Formal Verification Report of Compound V3 (Comet)

Summary

This document describes the specification and verification of Compound's Comet protocol using the Certora Prover. The work was undertaken from Feb 1st to March 24th, 2022, while the code was still in development. The latest commit reviewed and run through the Certora Prover was 4d1c1a42fc31b4b26129fe79e3d97ef637da9520.

The scope of this verification is Compound's Comet protocol which includes the following contracts:

- `Comet.sol`
- `CometExt.sol`

And their parent contracts (see Comet's architecture graph):

- `CometConfiguration.sol`
- `CometCore.sol`
- `CometFactory.sol`
- `CometMath.sol`
- `CometStorage.sol`

The Certora Prover proved that the implementation of Compound's Comet is correct with respect to formal specifications written by the Compound and Certora teams. Certora also performed a manual audit of these contracts.

During this verification process, the Certora Prover discovered bugs in the code which are listed in the tables below. All issues were promptly corrected by Compound, 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.

All the rules and specification files are publicly available and can be found in Compound's Comet repository.

List of Main Issues Discovered

Severity: Criticial

Issue:	**Possible liquidation of a collateralized account**
Description:	`updateAssetIn()` and `isInAsset()` use a mask of size `uint8` to update and retrieve data from the bit vector `assetsIn` . The bit vector's size is 16 bits, which renders the mask too small to update and retrieve data correctly for the assets that are in positions 8-15. For example, for an asset with `offset = 10` , `uint8(1) << 10 = 0` meaning the required mask won't be used and a 0 mask will be used instead. As a result `assetsIn` will be updated incorrectly, and the collateral provided will not be counted. The same problem is present in `isInAsset()` .
Property Violated:	Integrity of update of userCollateral (property #37)
Compound Response:	This issue was fixed in commit 4cbb7e6a3338a288bb95ba4992bd180dcd3721d6.

Severity: Critical

Issue:	**Creating collateral from nothing by using `transferCollateral`**
Description:	When calling `transferCollateral()` the new balances of the sender and the recipient are calculated and stored in two steps. Since the calculation is being done first for both users, it is possible to create funds from nothing by specifying `src = dst` . In this case the increment of `dst` balance overwrites the decrement of the `src` balance, effectively increasing the balance by the amount given.
Properties Violated:	• Total collateral per asset (property #1) • Verify transferAsset (property #12)
Compound Response:	This issue was fixed in commit c0d8a11f424747204ce680f0fe17441368f4d85c.

Severity: Critical

Issue:	**Creating base token from nothing by using `transferBase`**

Issue:	Creating base token from nothing by using `transferBase`
Description:	When calling `transferBase()` the new balances of the sender and the recipient are calculated and stored in two steps. Since the calculation is being done first for both users, it is possible to create funds from nothing by specifying `src = dst`. In this case the increment of `dst` balance overwrites the decrement of the `src` balance, effectively increasing the balance by the amount given.
Properties Violated:	• Total base token (property #5) • Verify transferAsset (property #12)
Compound Response:	This issue was fixed in commit c0d8a11f424747204ce680f0fe17441368f4d85c.

Severity: Critical

Issue:	Wrong calculation of `principalValue`
Description:	The function `principalValue()` is using the functions `presentValueSupply()` and `presentValueBorrow()` to calculate the principle value. It should use `principalValueSupply()` and `principalValueBorrow()` instead.
Property Violated:	--
Compound Response:	This issue was fixed in commit ffac97079f6573cc7cdb8ebc4e024ffce55825e9.

Severity: Medium

Issue:	Incorrect liquidation computation
Description:	When invoking `absorbInternal()`, `accrue()` is being called after the check `isLiquidatable()`. In practice, it means that the check whether a user is liquidatable is being done on a non-updated state of the system, i.e. on the state of the user from the last time an update was called. For example, a borrower can be not liquidatable at time $t=0$, then after some time (t) passes and debts accumulate, the borrower enters a liquidatable state at time t. If at no point in time $t'\geq t$ did anybody call the `accrue()` function through a financial action-- `withdraw()`, `supply()`, `transfer()` --a call to `absorbInternal()` will not allow users to absorb the borrower's assets even though in reality he/she should be liquidatable.

Issue:	Incorrect liquidation computation
Property Violated:	--
Compound Response:	This issue was fixed in commit cf066c4995162d5ac7d455a33e442d8dc7cbb2bb.

Severity: Medium

Issue:	Incorrect gain of assets, incorrect option to `buyCollateral`
Description:	`withdrawReserves()` and `buyCollateral()` use the `getReserves()` function to check the present value of `totalSupply` and `totalBorrow` . If `accrue()` is not called beforehand, the present values may not be fully up-to-date. As a result the calculated reserves amount will be inaccurate. This will prevent the governor from taking out its rightful assets, or mean that `getReserves()` retrieves a larger number than the governance. In addition, one might be able to buy collateral (at a discount) when not appropriate.
Property Violated:	Balance change vs accrue (property #9)
Compound Response:	This issue was fixed in commit 59def475c9ca9570f690201b7dd07a3ce1ed1a6b.

Severity: Low

Issue:	Incorrect collateral representation
Description:	`absorbInternal()` sets all of a user's collateral assets to 0, but never updates the user's `assetIn()` . This means that even though the collateral balance of a user is 0, the bit is still 'on' such that the function `isInAsset()` will return true. This wastes gas in methods like `isBorrowCollateralized()` , `getBorrowLiquidity()` , `isLiquidatable()` , `getLiquidationMargin()` , and `absorbInternal()` which count on the correct update of `assetIn` when iterating over collateral assets.
Property Violated:	AssetIn initialized with balance (property #8)
Compound Response:	This issue was fixed in commit 83211fa995a1e3bb79d98cbfdd453f0ce7f7b2e7.

Gas Optimization

- **`accrue()` can be called in `absorb()` instead of `absorbInternal()`** - Currently `accrue()` is being called in `absorbInternal()` and therefore being called in every loop iteration over the array of accounts. The accrual can be moved to `absorb()` to save some unnecessary operations.

- **Update of accrual time can be saved in some cases** - The update `lastAccrualTime = now_` in `accrue()` can be done inside the `if (timeElapsed > 0)` to save gas on storage.

- **Redundant use of `Safe64()`** - The use of `Safe64()` on `asset.scale` in `isBorrowCollateralized` and `getBorrowLiquidity` is redundant since scale is already a `uint64`.

- **Redundant assignment of `TotalsCollateral` to memory** - In `withdrawCollateral()` there is an assignment of `totalsCollateral` into a local variable which is redundant as it's accessed only once throughout the method.

- **Redundant check in `absorbInternal()`** - In `absorbInternal()` there is no need to check `if seizeAmount > 0`, because of the `isInAsset()` check beforehand.

- **Redundant assignment of `newBalance` in `absorbInternal()`** - In `absorbInternal()`, a more efficient way to execute the line `newBalance = newBalance < 0 ? int104(0) : newBalance` is by replacing it with `if(newBalance < 0) { newBalance = 0}`.

All the gas optimization suggestions were implemented in commit 10ca0422e4e983d8384a08c5d19ecb34515b66aa.

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.

Summary of Formal Verification

Overview of Comet Protocol

Compound Comet is a protocol that allows supplying volatile assets as collateral, and borrowing only a single (e.g. stable) coin.

The system pre-defines 2 sorts of tokens:

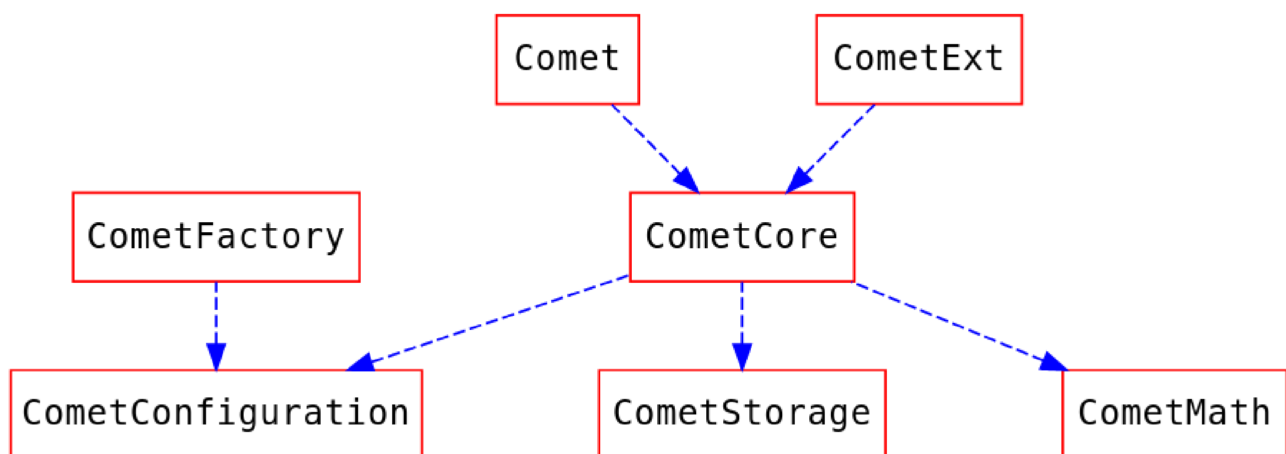
- **Base Token** - A single token, most likely a stable coin, that can be borrowed from or supplied to the system with interest determined by parameters of the system.
- **Collateral Token** - Up to 16 distinct ERC20 assets, most likely volatile coins/tokens, that can be supplied as collateral against a loan.

The protocol has a distinct borrow and supply rate curve for the borrowed token.

In the event a borrower's collateral does not cover its current debt, a 2 step absorption mechanism is provided. Absorption can be executed by anyone:

- **Step 1** - Once a user (Alice) is in a liquidatable state, any user in the system (including Alice herself) can call `absorb()` on her. The absorption sorts out Alice's balance sheet by giving up her debts while absorbing all of her collateral assets **to the system**. The user that successfully called `absorb()` receives `LiquidatorPoints` .
- **Step 2** - Now that Alice's account is balanced, and her collateral has been absorbed into the system, the collateral is available for anyone to buy for a discount.

Compound Comet Architecture



As the goal of Comet is to be highly optimized for a particular use case, it seeks to minimize the number of contracts involved. The protocol is implemented primarily in a monolithic contract.

Contract Functions

Supply

Serves three distinct purposes:

1. Supply liquidity in Base tokens
2. Supply collateral in volatile assets
3. Return borrowed Base tokens

Withdraw

Serves three distinct purposes:

1. Withdraw liquidity in Base tokens
2. Withdraw collateral
3. Borrow Base tokens

getSupplyRate

Returns the actual interest rate promised to liquidity providers. The rate is a function of liquidity utilization, interest rate slope and a reserve scaling factor.

getBorrowRate

Returns the actual interest rate the borrowers should expect. The rate is a function of liquidity utilization and interest rate slope.

getUtilization

Returns the rate of the utilization, or the rate of efficiency in utilizing the liquidity.

getReserves

Returns the amount of reserves left in the system. The protocol designates a part of the supplied liquidity and a part of the accumulated interest to the reserves. These reserves are then used in special cases, for example when the borrower fails to comply with the collateral obligations and the loan has to be liquidated.

withdrawReserves

Withdraws remaining base token reserves, if called by the governor.

isBorrowCollateralized

Recalculates the present value of the borrower's collateral and the present value of the borrowed amount then evaluates them with respect to the protocol's collateralization policy.

isLiquidatable

Recalculates the present value of the borrower's collateral and the present value of the borrowed amount then evaluates them with respect to the protocol's liquidation policy.

absorb

Liquidates a loan and transfers the remaining collateral to the contract. A user may call absorb if `isLiquidatable = true` .

buyCollateral

Allows a user to buy the "absorbed" collateral from the system by providing the appropriate amount of Base tokens.

transferAsset

Allows user A to transfer his assets stored in the system to user B so long as those assets do not serve as collateral.

accrueInternal (internal function)

An important function that updates the aggregated interest rates for suppliers and for the borrowers.

Verified Contracts

In order to have a better coverage of the system, and due to computational complexities that arise in the process of verifying a system as complex as Comet, some harnesses to the original contract were required.

The harnesses contain additional or modified functionalities for the original contract.

We've split our harnesses into several levels of modifications, which allows for greater control in determining which rule is verified under which conditions and assumptions. All harnesses inherit from the original contract they modify.

```
digraph HarnessStructure {
graph [label="Harness Structure\n\n", labelloc=t, fontsize=25];
nodesep=1.0
node [color=Red,fontname=Courier,shape=box]
edge [color=Blue, style=dashed]

{ rank = sink; Comet; }
CometHarnessGetters;
{ rank = source CometHarnessWrappers; }
```

```
{ rank = source; CometHarness; }

Comet->{CometHarnessGetters} [dir=back]
CometHarnessGetters->{CometHarnessWrappers CometHarness}[dir=back]
}
```

CometHarnessGetters

A collection of getters to access specific fields in a given struct that otherwise would be invisible to the SPEC file.

CometHarnessWrappers

A collection of functions that wrap around source code functions and data structures that otherwise would be invisible to the SPEC file. (for example: internal functions).

CometHarness

A collection of summarizations and simplifications of methods and components of Comet.

Some summarizations are broadly safe approximations as important properties have been proven on the original code. For example: replacing the extensively bitwise operation functions `isInAsset` with a simpler but semantically equivalent summarization. Since `isInAsset` has been proven to be correct with respect to a set of properties, this simplification is a safe approximation for use without additional constraints.

In contrast, other included simplifications are under-approximations as they don't take into account all possible states of the contract. These simplifications are useful for providing additional coverage of special cases, but associated properties should be considered as verified only under certain conditions and assumptions, as noted in the sections below.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

- When verifying contracts that make external calls, we assume that those calls can have arbitrary side effects outside of the contracts, but that they do not affect the state of the contract being verified. This means that some reentrancy bugs may not be caught.
- We unroll loops. Violations that require a loop to execute more than twice will not be detected.

Some rules are proven over the following set of simplified assumptions:

- The `accrue` method does not change any value - specifically the supply and borrow rate
- The supply and borrow rate are kept constant at the initial value
- Base scale set to Factor scale
- `AccrualDescaleFactor` and `trackingIndexScale` are both 1

Notations

✔️ indicates the rule is formally verified on the latest reviewed commit. We write ✔️ * when the rule was verified on the simplified assumptions described above in "Assumptions and Simplifications Made During Verification".

❌ indicates the rule was violated under one of the tested versions of the code.

🔁 indicates the rule is timing out.

Our tool uses 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. This logical structure is reflected in the included formulae for many of the properties below. Note that p and q here can be thought of as analogous to require and assert in Solidity.

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 Comet

Over fifty properties are defined for verification with Comet. The properties are separated into sections, starting with the high-level properties.

In addition, a methodology for checking reentrancy safety has been applied to verify the unlikely case of a call back from an ERC20 contract.

Lastly, a set of properties for listing ERC20 assets are formally defined and can be easily applied on any ERC20 contract.

High Level Properties of the System

files: comet.spec, cometTotalsAndBalances.spec

1. **Total collateral per asset** ✔ * - (❌ failed on older version of the code - issue 1) The sum of collateral per asset over all users must equal the total collateral for that asset

```
   sum(userCollateral[user][asset].balance) =
totalsCollateral[asset].totalSupplyAsset
```

2. **Total asset collateral vs asset balance** ✔ * The total supply of an asset must not exceed the contract's balance for that asset.

```
   totalsCollateral[asset].totalSupplyAsset ≤ asset.balanceOf(this)
```

3. **Base balance vs totals** ✔ * The base token balance of the system must be at least as large as the total supply of base tokens minus the total base tokens borrowed

```
   baseToken.balanceOf(currentContract) ≥ getTotalSupplyBase() −
getTotalBorrowBase()
```

4. **Collateral totalSupply LE supplyCap** ✔ * The total supply of an asset must not be greater than its supply cap

```
   totalsCollateral[asset].totalSupplyAsset ≤
getAssetSupplyCapByAddress(asset)
```

5. **Total base token** ✔ * (❌ failed on older version of the code - issue 5) The sum of principal balances over all users must equal the total base token within the system

```
   sum(userBasic[user].principal) == totalsBasic.totalSupplyBase −
totalsBasic.totalBorrowBase
```

6. **Balance change by allowed only** ✔ * A user's principal balance may decrease only by a call from them or from a permissioned manager

```
   {
       x = userBasic[user].principal ∧
       b = userCollateral[user][asset].balance ∧
       p = hasPermission[user][msg.sender]
   }

       < call op() by msg.sender>

   {
       ( userBasic[user].principal < x  ⇒ ( user = msg.sender v p ) ) ∧
       userCollateral[user][asset].balance < y ⇒ ( user = msg.sender v p
```

```
v op=absorb )
    }
```

7. **Collateralized after operation** ✔ * Any operation on a collateralized account must leave the account collateralized

```
{
    isBorrowCollateralized(t, user)
}

    < call op() at time t >

{
    isBorrowCollateralized(t, user)
}
```

8. **AssetIn initialized with balance** ✔ * (❌ failed on older version of the code - issue 7) The assetIn switch of a specific asset must be initialized along with the collateral balance

```
userCollateral[user][asset].balance > 0 ⇔
    isInAsset(userBasic[user].assetsIn, asset.offset)
```

9. **Balance change vs accrue** ✔ * (❌ failed on older version of the code - issue 6) Base balance must change only on updated accrued state

```
{
    balance_pre = tokenBalanceOf(_baseToken,currentContract)
}

    < call any function >

{
    balance_pre ≠ tokenBalanceOf(_baseToken,currentContract) ⇒
            accrueWasCalled()
}
```

10. **Balance change vs registered** ✔ * A change in the system's balance of some asset must only occur for assets registered as recognized assets

```
{
    registered = getAssetInfoByAddress(token).asset == token ∧
    token ≠ _baseToken ∧
    b = tokenBalanceOf(token,currentContract)
}
```

```
        < call any function >

    {
        tokenBalanceOf(token,currentContract) ≠ b ⇒ registered
    }
```

11. **Usage registered assets only** ✔ * Every function that has an asset argument must revert on a non-registered asset.

```
    {

    }

        < call any function with asset >

    {
        getAssetInfoByAddress(asset).asset == asset
    }
```

12. **Verify transferAsset** ✔ * (❌ failed on older version of the code - issue 1,2) A transfer must not change the combined presentValue of src and dst

```
    {
        p = baseBalanceOf(src) + baseBalanceOf(dst) ∧
        c = getUserCollateralBalance(asset, src) +
getUserCollateralBalance(asset, dst)
    }

        transferAssetFrom(src, dst, asset, amount)

    {
        baseBalanceOf(src) + baseBalanceOf(dst) = p ∧
        getUserCollateralBalance(asset, src) +
getUserCollateralBalance(asset, dst)  = c
    }
```

Properties Regarding Withdrawal and Supply

files: cometAbsorbBuyCollateral.spec

13. **Withdraw reserves decreases** ✔ * When a manager withdraws from the reserves, the system's reserves must decrease

```
    {
        before = getReserves()
    }
```

```
    withdrawReserves(to,amount)

{
    amount > 0 ⇒ getReserves() < before
}
```

14. **Withdraw reserves monotonicity** ✔ * The larger a withdrawal from reserves a manager makes, the smaller the remaining reserves in the system must be

```
{

}

    withdrawReserves(x); r1 = getReserves()
    ~
    withdrawReserves(y); r2 = getReserves()

{
    x > y ⇒ r1 < r2
}
```

15. **Supply increase balance** ✔ * If a certain amount of an asset is supplied, the balance of that asset must increase by that amount

```
{
    balance1 = tokenBalanceOf(asset, currentContract)
}

    supply(asset, amount)

{
    tokenBalanceOf(asset, currentContract) - balance1 == amount
}
```

16. **Withdraw decrease balance** ✔ * If a certain amount of an asset is withdrawn, the balance of that asset must decrease by that amount

```
{
    b = tokenBalanceOf(asset, currentContract)
}

    withdraw(asset, amount)

{
    b - tokenBalanceOf(asset, currentContract) == amount
}
```

17. **Additivity of withdraw** ✔ * Performing two distinct withdrawals must result in the same outcome as performing a single withdrawal for the same total amount

```
    {

    }

        withdraw(Base, x); withdraw(Base, y) ; base1 :=
baseBalanceOf(e.msg.sender)
        ~
        withdraw(_baseToken, x + y); base2 := baseBalanceOf(e.msg.sender)

    {
        base1 == base2
    }
```

Properties Regarding absorb and buyCollateral

files: cometAbsorbBuyCollateral.spec

18. **Anti monotonicity of buyCollateral** ✔ * After a call to buy collateral: (i) the collateral balance must decrease (ii) the Base balance must increase (iii) the Base balance must increase if and only if the collateral balance decreases

```
    {
        balanceAssetBefore = tokenBalanceOf(asset, currentContract)     ∧
        balanceBaseBefore = tokenBalanceOf(_baseToken, currentContract)
    }

        buyCollateral(asset, minAmount, baseAmount, recipient)

    {
        tokenBalanceOf(asset, currentContract) ≤ balanceAssetBefore
∧
        balanceBaseBefore ≤ tokenBalanceOf(_baseToken, currentContract)
∧
        ( balanceBaseBefore < tokenBalanceOf(_baseToken, currentContract) ⇔
            tokenBalanceOf(asset, currentContract) < balanceAssetBefore )
    }
```

19. **BuyCollateral max** ✔ * When absorb is called and a user's collateral is added to the contract's collateral, a subsequent collateral purchase must not exceed the contract's collateral

```
{
    max = getUserCollateralBalance(currentContract, asset)
    balanceAssetBefore = tokenBalanceOf(asset, currentContract)
}

    buyCollateral(asset, minAmount, baseAmount, recipient)

{
    tokenBalanceOf(asset, currentContract) ≥ balanceAssetBefore − max
}
```

20. **Cannot absorb same account** ✔ * If the array of accounts has the same account twice then absorb must revert

```
{
    accounts[0] == account ∧ accounts[1] == account
}

    absorb@withrevert(absorber, accounts)

{
    lastReverted
}
```

21. **Absorb reserves decrease** ✔ * After absorption of an account, the system's reserves must not increase

```
{
    pre = getReserves()
}

    absorb(absorber, accounts)

{
    getReserves() ≤ pre
}
```

22. **Anti monotonicity of absorb** ✔ * On absorb, as the collateral balance increases the total BorrowBase must decrease

```
{
    balanceBefore = getUserCollateralBalance(this, asset),
    borrowBefore = getTotalBorrowBase()
}

    absorb(absorber, accounts)
```

```
{
    getUserCollateralBalance(this, asset) > balanceBefore ⇒
        getTotalBorrowBase() < borrowBefore
}
```

23. **Cannot double absorb** ✓ * The same account cannot be absorbed repeatedly

```
{

}
    absorb(absorber, accounts);
    absorb@withrevert(absorber, accounts)

{
    lastReverted
}
```

Properties Regarding Setting Allowance

files: cometExt.spec

24. **Allowance only zero or max** ✓ Spender's allowance must be equal to either 0 or to max_uint256

```
  allowance[owner][spender] == 0 v allowance[owner][spender] ==
max_uint256
```

25. **Approve fails on invalid allowance** ✓ Trying to approve an allowance which is not 0 or max_uint must fail

```
{
    0 < amount < max_uint256
}

    approve(spender, amount)

{
    lastReverted
}
```

26. **Valid allowance changes** ✓ Allowance must change only as a result of approve(), allow() and allowBySig(). Allowance must change for non msg.sender only as a result of allowBySig()

```
{
    allowanceBefore = allowance[owner][spender]
}

    < call any function f >

{
    allowanceAfter = allowance[owner][spender] ∧
    allowanceBefore ≠ allowanceAfter ⇒
        ( f == approve v f == allow v f == allowBySig )
}
```

27. **Valid approve succeeds** ✔ Approve with a valid amount (0 or max_uint256) must succeed

```
{
    amount = 0 v
    amount = max_uint256
}

    approve(spender, amount)

{
    ¬lastReverted
}
```

Properties Regarding Governance Methods

files: governance.spec, pause.spec, pauseGuardians.spec

28. **Methods pause and withdrawReserves by allowed only** ✔ Methods pause and withdrawReserves must be called only by governor or by pauseGuardian

```
{

}

    < call to pause() or withdrawReserves() >

{
    ¬lastReverted ⇒ (msg.sender = governor) v
                    (msg.sender = pauseGuardian)
}
```

29. **Ability to updates flag** ✔ Method pause must revert if and only if the sender is neither governor nor pause guardian

```
{

}

    pause()

{
    lastReverted ⇔ (msg.sender ≠ governor) ∧
                   (msg.sender ≠ pauseGuardian)
}
```

30. **Integrity of flag updates** ✔ After an update the getters must retrieve the same values as the arguments to pause

```
{

}

    pause(supplyPaused, transferPaused, withdrawPaused, absorbPaused,
buyPaused)

{
    ¬lastRevert ⇒ ( supplyPaused = supplyPaused() ∧
                    transferPaused = isTransferPaused() ∧
                    withdrawPaused = isWithdrawPaused() ∧
                    absorbPaused = isAbsorbPaused() ∧
                    buyPaused = isBuyPaused() )
}
```

31. **Pause supply functions** ✔ Supply functions must revert if pauseSupply is true

```
{
    flagSupply = get_supply_paused()
}

    < call any supply function >

{
    flagSupply ⇒ lastReverted
}
```

Similarly, the following properties are also defined:

32. **Pause transfer functions** ✔

33. **Pause withdraw functions** ✔

34. **Pause absorb** ✔

35. **Pause buyCollateral** ✔

Properties Regarding Asset Information

files: assetInfo.spec, userAssetIn.spec,

36. **Reversibility of packing** ✔ Unpacking assetInfo after packing must return the same info

```
getAssetInfo(getPackedAsset(assetInfo_struct)) == assetInfo_struct
```

37. **Integrity of update of userCollateral** ✔ (❌ failed on older version of the code - issue 4) If a specific asset balance is being updated from 0 to non-0 or vice versa, isInAsset must return the appropriate value

```
    {

    }

        updateAssetsIn(account, asset, initialUserBalance,
finalUserBalance);
        flagUserAsset_ := isInAsset(userBasic[user].assetsIn,
asset.offset);

    {
        ((initialUserBalance == 0 ∧ finalUserBalance > 0) ⇒ flagUserAsset_
) ∧
        ((initialUserBalance > 0 ∧ finalUserBalance == 0) ⇒ ¬flagUserAsset_
)
    }
```

38. **No change to other asset** ✔ Update assetIn must only change a single bit - two distinct asset bits must not change with the same call to update

```
    {
        flagUserAsset1 = isInAsset(userBasic[user].assetsIn, assetOffset1)
∧
        flagUserAsset2 = isInAsset(userBasic[user].assetsIn, assetOffset2)
∧
        assetOffset1 ≠ assetOffset2
    }

        updateAssetsIn(account, asset, initialUserBalance,
finalUserBalance)

    {
        _flagUserAsset1 = isInAsset(userBasic[user].assetsIn, assetOffset1)
```

```
v
        _flagUserAsset2 = isInAsset(userBasic[user].assetsIn, assetOffset2)
    }
```

39. **No change to other user's asset info** ✔ Update assetIn must change the assetIn of a single user - no other users may be affected by update

```
    {
        other ≠ user ∧
        assetIn = userBasic[other].assetsIn
    }

        updateAssetsIn(account, asset, initialUserBalance, 
finalUserBalance)

    {
        userBasic[other].assetsIn ≠ assetIn
    }
```

Properties Regarding Interest Computation in Internal Functions

files: interestComputations.spec, timeouts.spec,

40. **Monotonicity of supplyIndex and borrowIndex** ✔ SupplyIndex and borrowIndex must only increase due to a call to accrue

```
    {
        supply_index = getBaseSupplyIndex() ∧
        borrow_index = getBaseBorrowIndex() ∧
    }

        accrueInternal();

    {
        getBaseSupplyIndex() ≥ supply_index ∧
        getBaseBorrowIndex() ≥ borrow_index
    }
```

41. **Min value of supplyIndex and borrowIndex** ✔ SupplyIndex and borrowIndex must be at least the minimum

```
BaseSupplyIndex() ≥ BaseIndexScale() ∧
    BaseBorrowIndex() ≥ BaseIndexScale()
```

42. **SupplyRate vs utilization** ✔ Utilization increase over time must imply supplyRate increase If the utilization is increased the supplyRate cannot decrease

```
utilization(t1) > utilization(t2) ⇒ supplyRate(t2) ≥ supplyRate(t1)
```

43. **Utilization zero**✔ When utilization is 0, borrow rate must equal the base borrow rate.

```
utilization(t) = 0 ⇒  getBorrowRate(t) = perSecondInterestRateBase()
```

44. **BorrowBase vs utilization** ✔ If nobody borrows from the system, the utilization must be 0

```
getTotalBorrowBase(t) = 0 ⇒ utilization(t) = 0;
```

45. **Valid change to isLiquidatable** ✔ When taking into account only time change, isLiquidatable must change from false to true only if getPrice() has changed for base or asset

```
t2 > t1 ∧ ¬isLiquidatable(t1,account) ∧ isLiquidatable(t1,account) ⇒
    ( getPrice(t1,priceFeedBase) ≠  getPrice(t2,priceFeedBase) ∨
         getPrice(e1,priceFeedAsset) ≠ getPrice(e2,priceFeedAsset) )
```

46. **Is collateralized vs liquidatable** ✔ If an account is collateralized then it must not be liquidatable

```
isBorrowCollateralized(account) ⇒ ¬isLiquidatable(account)
```

47. **PresentValue vs principalValue** ✔ PresentValue must always be greater than principalValue

```
principalValue = principalValue(presentValue) ⇒
    ((presentValue ≥ 0 ⇒ presentValue ≥ principalValue) ∧
        (presentValue < 0 ⇒ presentValue ≤ principalValue))
```

48. **Present value more than zero** ✔ PresentValue must be positive if and only if principalValue is positive

```
( principalValue = principalValue(presentValue) ∧
    presentValue = presentValue(principalValue) ) ⇒
    ( presentValue > 0 ⇔ principalValue > 0 )
```

49. **Utilization zero** ✔ If utilization is 0, then supplyRate must be 0

```
Utilization == 0 ⇒ SupplyRate == 0
```

50. **SupplyRate revert characteristic** ✔ GetSupplyRate must always revert if reserveRate > FACTOR_SCALE

```
{

}

    getSupplyRate()

{
    reserveRate > FACTOR_SCALE ⇒ lastReverted
}
```

Verification of Reentrancy Safety

To verify the safety for re-entrency calls from a possible malicious/broken ERC20, we prepared an ERC20 token that callsback Comet functions. We allowed the ERC token to have a call back to the following Comet functions:

- buyCollateral
- supply
- withdraw
- transferAssetFrom
- transferFrom

We have checked properties 1- 23 given this malicious token as a potential asset.

As expected, a few properties do not hold but do not indicate an issue in the Comet code. For example, property #6 (Balance change by allowed only) fails when the callback is to withdraw since the Certora prover considers the case where the ERC20 asset has an allowance.

In addition, some properties timed out and could not be verified.

Formal Properties for ERC20 Assets to be Listed

As part of our effort to secure the protocol, we've prepared a scaffold specification for ERC20 tokens. This set of rules is aimed as a preemptive measure for verifying certain desired properties in an ERC20 token before enlisting it in the system. In essence, the specifications can be thought of as an inspection tool to ascertain whether specified tokens meet or violate the given properties.

We present here a set of properties we've written for that purpose. This is a framework on which the community can build, solidify, and improve their set of desired properties.

1. **noFeeOnTransferFrom** Verify that there is no fee on transferFrom() (like potentially on USDT)

```
{
    balances[bob] = y
    allowance(alice, msg.sender) ≥ amount
}

    transferFrom(alice, bob, amount)

{
    balances[bob] = y + amount
}
```

2. **noFeeOnTransfer** Verify that there is no fee on transfer() (like potentially on USDT)

```
{
    balances[bob] = y
    balances[msg.sender] ≥ amount
}

    transfer(bob, amount)

{
    balances[bob] = y + amount
}
```

3. **transferCorrect** Verify that token transfer works correctly. Balances are updated if not reverted. If reverted then the transfer amount was too high, or the recipient is 0.

```
{
    balanceFromBefore = balanceOf(msg.sender)
    balanceToBefore = balanceOf(to)
}

    transfer(to, amount)
```

```
{
    lastReverted ⇒ to = 0 ∨ amount > balanceOf(msg.sender)
    ¬lastReverted ⇒ balanceOf(to) = balanceToBefore + amount ∧
                    balanceOf(msg.sender) = balanceFromBefore − amount
}
```

4. **TransferFromCorrect** Verify that transferFrom works correctly. Balances are updated if not reverted. If reverted, it means the transfer amount was too high, or the recipient is 0

```
{
    balanceFromBefore = balanceOf(from)
    balanceToBefore = balanceOf(to)
}

    transferFrom(from, to, amount)

{
    (lastreverted ⇒ to = 0 ∨ amount > balanceOf(from)) ∧
    (¬lastreverted ⇒ (balanceOf(to) = balanceToBefore + amount ∧
                      balanceOf(from) = balanceFromBefore − amount))
}
```

5. **TransferFromReverts** Verify that transferFrom reverts if and only if the amount is too high or the recipient is 0.

```
{
    allowanceBefore = allowance(alice, bob)
    fromBalanceBefore = balanceOf(alice)
}

    transferFrom(alice, bob, amount)

{
    lastReverted ⇔ allowanceBefore < amount ∨ amount >
fromBalanceBefore ∨ to = 0
}
```

6. **ZeroAddressNoBalance** Verify that the balance of address 0 is always 0

```
balanceOf[0] = 0
```

7. **NoChangeTotalSupply** Verify that contract calls don't change token total supply.

```
{
    supplyBefore = totalSupply()
}

    < call any function >

{
    supplyBefore = totalSupply()
}
```

8. **ChangingAllowance** Verify that allowance changes correctly as a result of calls to approve, transfer, increaseAllowance, decreaseAllowance

```
{
    allowanceBefore = allowance(from, spender)
}

    < call any function f>

{
    f = approve(spender, amount) ⇒ allowance(from, spender) = amount
    f = transferFrom(from, spender, amount) ⇒ allowance(from, spender)
= allowanceBefore − amount
    f = decreaseAllowance(spender, delta) ⇒ allowance(from, spender) =
allowanceBefore − delta
    f  = increaseAllowance(spender, delta) ⇒ allowance(from, spender) =
allowanceBefore + delta
    other f ⇒ allowance(from, spender) == allowanceBefore
}
```

9. **TransferSumOfFromAndToBalancesStaySame** Verify that transfer from a to b doesn't change the sum of their balances

```
{
    balancesBefore = balanceOf(msg.sender) + balanceOf(b)
}

    transfer(b, amount)

{
    balancesBefore = balanceOf(msg.sender) + balanceOf(b)
}
```

10. **TransferFromSumOfFromAndToBalancesStaySame** Verify that a transfer using transferFrom() from a to b doesn't change the sum of their balances

```
{
    balancesBefore = balanceOf(a) + balanceOf(b)
}

    transferFrom(a, b)

{
    balancesBefore = balanceOf(a) + balanceOf(b)
}
```

11. **TransferDoesntChangeOtherBalance** Verify that transfer from msg.sender to alice doesn't change the balance of other addresses

```
{
    balanceBefore = balanceOf(bob)
}

    transfer(alice, amount)

{
    balanceOf(bob) = balanceBefore
}
```

12. **TransferFromDoesntChangeOtherBalance** Verify that transfer from alice to bob using transferFrom doesn't change the balance of other addresses

```
{
    balanceBefore = balanceOf(charlie)
}

    transferFrom(alice, bob, amount)

{
    balanceOf(charlie) = balanceBefore
}
```

13. **OtherBalanceOnlyGoesUp** Verify that the balance of an address, who is not a sender or a recipient in transfer functions, doesn't decrease as a result of contract calls

```
{
    balanceBefore = balanceOf(charlie)
}

    < call any function f >

{
    f ≠ transfer ∧ f ≠ transferFrom ⇒ balanceOf(charlie) =
```

```
balanceBefore
    }
```

The Certora team added a demo on 3 common tokens deployed on mainnet - USDC, Sushi and FTT. The spec doesn't pass completely on any of these tokens, because each of them has some functions that violate the rules: mint/burn, pause/blacklist, and others.