Formal Verification Report of Gsm

Summary

This document describes the specification and verification of GHO Stability Module using the Certora Prover. The work was undertaken from August 9, 2023 to December 7, 2023. The latest commit that was reviewed and run through the Certora Prover is f368bef

The scope of our verification includes the following contracts:

- Gsm.sol
- Gsm4626.sol
- FixedPriceStrategy.sol
- FixedPriceStrategy4626.sol
- FixedFeeStrategy.sol
- OracleSwapFreezer.sol

The Certora Prover proved the implementation correct with respect to the formal rules written by the Certora team. During verification, the Certora Prover discovered bugs in the code which are listed in the tables below. All issues were promptly addressed. The fixes were verified to satisfy the specifications up to the limitations of the Certora Prover.

List of Main Issues Discovered

Standard GSM Issues

Issue:	`getAssetAmountForSellAsset` is slightly unfair to user
Severity:	**Informational**

Issue:	`getAssetAmountForSellAsset` is slightly unfair to user
Violated property:	**getAssetAmountForSellAsset_optimality**
Description:	When user wants to swap assets for `x` GHO, `getAssetAmountForSellAsset(x)` should report, among other values, the amount of assets to sell. In some cases the system recommends selling more asset (and receive more GHO) than necessary, i.e., the system encourages users to spend more assets than needed. The recommended amount might also result in higher percentual fees than the real minimum. The error can be in the range of `10^-17` GHO.
Example:	`priceRatio = 1`, `sellFee = 49%`. `getAssetAmountForSellAsset(36) = (x=71, 71, 35)`. When calling `sellAsset(x, _, _)` they receive `36*10^-18` GHO for any `70 <= x <= 71`.
Mitigation/Fix:	Fixed in PR#369

Issue:	Inconsistency in the amount of GHO user asks to sell and how much GHO is actually deducted from their account.
Severity:	**Informational**
Violated property:	**R4_sellGhoUpdatesAssetBuyerGhoBalanceGe**
Description:	When swapping GHO for underlying asset, gsm requires user to specify an amount of underlying asset `asset_amount` they would like to get in `buyAsset(asset_amount)`. The function computes the amount of GHO needed to be sold in order to acquire the desired amount of asset. The API does not provide a function for buying underlying asset that takes the amount of GHO to be sold. For the case where a user wants to sell a specific amount of GHO, `gho_amount`, the contract provides a view function, `getAssetAmountForBuyAsset(gho_amount)`, which supposedly returns the amount of assets that needs to be passed to `buyAsset()` in order to sell exactly `gho_amount`. Since the precision (number of decimals) of the asset is fixed, it is possible that there is no amount of GHO that would correspond to a given `asset_amount`. In these cases gsm behaves inconsistently: for some values, gsm charges more than user specifies and for some others, less.

Issue:	Inconsistency in the amount of GHO user asks to sell and how much GHO is actually deducted from their account.
Example:	• Let `gho_amount = 6`, price ratio `PR = 4`, underlying asset units `UAU = 1`, buy fee in BP `buyFeeBP = 0`. The change in GHO balance after `buyAsset` is 8, which is greater than `gho_amount`. • Let GHO amount `gho_amount = 3*10^36+5`, price ratio `PR = 1*10^36+2`, underlying asset units `UAU = 1`, buy fee in BP `buyFeeBP = 0`. The change in GHO balance after `buyAsset` is `2*10^36+4`, which is less than `gho_amount`
Mitigation/Fix:	Fixed in PR#369

Issue:	`getAssetAmountForBuyAsset` exceeds user-given bound
Severity:	**Informational**
Violated property:	**R1_getAssetAmountForBuyAssetRV2**
Description:	The user may ask the amount of assets `a` to provide for `buyAsset(a)` by calling `getAssetAmountForBuyAsset(max)`, where `max` is the maximum amount of GHO user is willing to pay. One of the return values of `getAssetAmountForBuyAsset` is the exact amount of GHO that will be deducted. This value can be higher than `max` by at most `2*10^-18` GHO.
Example:	`priceRatio = 1`, `buyFee = 25.01%`, `getAssetAmountForBuyAsset(4) = (2,6,4,2)`, i.e., user wants to spend at most `4*10^-18 GHO`, gsm tells him they should buy 2 assets and they will pay `4*10^-18` GHO + `2*10^-18` GHO fee.
Mitigation/Fix:	Fix rounding directions in fee strategy. Fixed in PR#369

Issue:	Collected buy fees are rounded down and can be 0 in extreme cases
Severity:	**Informational**
Violated property:	**collectedBuyFeeIsAtLeastAsRequired**

Issue:	Collected buy fees are rounded down and can be 0 in extreme cases
Description:	In extreme cases, for sufficiently small amounts of bought asset, the fee collected by the contract can be zero even if the fee expressed in basic points is non-zero.
Example:	`buyFee = 0.02%`, `underlyingAssetDecimals = 10^13`, `price_ratio = 14,995,000,000,000,001`. `BuyAsset(minAmount=3)` → `_calculateGhoAmountForBuyAsset(3)` → `(2, 3001, 3001, 0)`.
Mitigation/Fix:	PR#369

Issue:	`getGrossAmountFromTotalSold` does not revert when `_sellFee = 100%`.
Severity:	**Informational**
Violated property:	**getGrossAmountFromTotalSold_isMonotoneInTotalAmount**
Description:	For `x != 0`, there's no correct return value of `getGrossAmountFromTotalSold(x)` with `_sellFee = 100%`. Returning `0` is arbitrary and may lead to unexpected behaviour on the side of the caller.
Example:	For `_sellFee = 100%`, `getGrossAmountFromTotalSold(10) = 0`. It is never possible to receive `10*10^-18` GHO for selling any amount of assets. With `_sellFee = 100%`, `sellAsset(x)` always provides `0` GHO.
Mitigation/Fix:	Prevent 100% sell fee. Fixed in PR#369

Issue:	`getAssetAmountForBuyAsset` provides incorrect information
Violated property:	**R2_getAssetAmountForBuyAssetRV_vs_GhoBalance**
Severity:	**Informational**
Description:	*Note: this issue is similar to the one called `getAssetAmountForBuyAsset` exceeds user-given bound, but on an earlier version of gsm*. The method `getAssetAmountForBuyAsset` informs the user how much asset they should buy in order to spend a specified amount of GHO. It can mislead the user by telling them they will be charged `x`

Issue:	`getAssetAmountForBuyAsset` provides incorrect information
	GHO while actually charging them `(x+1)`. I.e., it can be off by at most 1 in favor of the protocol.
Example:	`priceRatio = 1`, `buyFee = 50%`, `getAssetAmountForBuyAsset(4) = (3, 3, 1)`, i.e., user wants to spend `4*10^-18` GHO, gsm tells they should buy 3 assets and they will pay `3*10^-18` GHO + `1*10^-18` GHO fee. When calling `buyAsset(3, _, _)` the user is charged `5*10^-18` GHO instead of `4*10^-18`.
Mitigation/Fix:	PR#369

Issue:	Inconsistency between the reported and accrued fees when swapping
Severity:	**Informational**
Violated property:	**R2_getAssetAmountForBuyAssetNeBuyAssetFee**, **R4_estimatedBuyFeeGeActualBuyFee**, **R3_estimatedSellFeeCanBeHigherThanActualSellFee**
Description:	When a swap takes place in gsm, the contract may collect a fee. The fee is represented in basic points. When a concrete transaction takes place the fee in basic points is used to obtain a concrete fee in GHO. The API exposes the fee in three different ways. (1) Directly based on BP through `getBuyFee(x)` and `getSellFee(x)`, (2) As the fee reported by `getAssetAmountForBuyAsset(x)` and `getAssetAmountForSellAsset(x)`, (3) As the fee accrued through `buyAsset(a)` and `sellAsset(a)`. The fee reported by `getBuyFee(x)` and `getSellFee(x)` can be less than, greater than, or equal to the fee accrued by `buyAsset(a)` and `sellAsset(a)`. In addition, the fee reported by `getAssetAmountForBuyAsset(x)` can be less than, greater than, or equal to the fee accrued by `buyAsset`.
Mitigation/Fix:	ections. Fixed in PR#369

Issue:	User may pay slightly more GHO than the maximum they requested
Severity:	**Informational**

Issue:	User may pay slightly more GHO than the maximum they requested
Violated property:	**R2_getAssetAmountForBuyAssetRV_vs_GhoBalance**
Description:	The user may ask the amount of assets `a` to provide for `buyAsset(a)` by calling `getAssetAmountForBuyAsset(max)`, where `max` is the maximum amount of GHO user is willing to pay. When the return value is provided to `buyAsset`, it is possible that the user is charged slightly more than `max` GHO.
Mitigation/Fix:	Fix rounding directions in fee strategy. Fixed in PR#369

Issue:	Bad rounding may steal small amounts of asset from the contract
Severity:	**Informational**
Violated property:	**totalAssetsNotIncrease**
Description:	User can sell GHO and get more underlying asset than they should due to a rounding error. As a result, a user can obtain assets from the system or get preferential treatment in comparison to other users. The value of stolen asset is at most `9*10^-19` GHO
Example:	Consider the following problem with the `buyAsset()` function. Let `UNDERLYING_ASSET_DECIMALS=19` (i.e., `_underlyingAssetUnits = 10^19`) and `PRICE_RATIO=10^18`. We assume zero fees. Then • `getAssetPriceInGho(amount=11)` returns `1` meaning that a user needs `1*10^-18` GHO to buy `11` underlying assets (since `floor(11*10^18/10^19) = 1`). However, the user can buy up to `19` underlying assets for `1*10^-18` GHO. • Consider `Alice` and `Bob`, both selling `10` underlying assets and obtaining `1*10^-18` GHO. After these transactions there are exactly `20` underlying assets in the system. `Alice` buys `19` assets for `1*10^-18` GHO. Then `Bob` cannot buy back `10` underlying assets for `1*10^-18` GHO since only `1` underlying asset remain in system.
Mitigation/Fix:	Fixed in PR#369

Issue:	Overbacking when selling asset.
Severity:	**Informational**
Violated property:	**systemBalanceStabilitySell**
Description:	When asset is sold for GHO, the value of the minted GHO is not ex equal to the value of the asset. The GHO minted may be smaller th asset value, which will result in overbacking.
Example:	Consider the following settings: `PRICE_RATIO=10^18`, `_underlyingAssetUnits = 10^15`, `PercentageMath.PERCENTAGE_FACTOR=10000` and `_sellFee=9984`. we call `sellAsset(maxAmount=1)`, from which `Gsm._calculateGhoAmountForSellAsset(assetAmount=1)` is calle Inside the function following computation happens: 1. `PriceStrategy.getAssetPriceInGho(assetAmount=1)` return `10^18/10^15 = 1000` (which is correct). 2. `FeeStrategy.getSellFee(grossAmount=1000)` returns `1000*9984/10000 = 998.4 ~= 999`, 3. `FeeStrategy.getGrossAmountFromTotalSold(totalAmount=(1 999))` returns `1*10000/(10000-9984) = 625` 4. `PriceStrategy.getGhoPriceInAsset(ghoAmount=625,roundUp` returns `625*10^15/10^18~=1`. Finally `_calculateGhoAmountForSellAsset(assetAmount=1)` retur `(1,1,625,624)`. So `625*10^-18` GHO is minted and user receive `1*10^-18` GHO selling `1` asset which is actually worth `1000*10^` GHO.
Mitigation/Fix:	PR#369

GSM4626 Issues

Issue:	Gsm4626: Breaking up transactions slightly decreases total cost.
Severity:	**Informational**
Description:	When buying assets user can pay slightly less by splitting the purchase into smaller transactions. To obtain meaningful financial gain, the number of transactions needs to be in the order of 10^19.

Issue:	Gsm4626: Breaking up transactions slightly decreases total cost.
Violated property:	**R1_getAssetAmountForBuyAssetRV2**
Example:	Using `previewRedeem(shares) = RoundDown((shares*5)/3)`, `previewWithdraw(shares) = RoundUp((shares*3)/5)`. `price ratio = 100000000000000002`, `buyFee = sellFee = 0`, `underlyingAssetDecimals = 17`, we have: `getAssetAmountForBuyAsset(43) = (3, 51, 51, 0)`, `getAssetAmountForBuyAsset(33) = (2, 31, 31, 0)`, `getAssetAmountForBuyAsset(11) = (1, 11, 11, 0)`
Mitigation/Fix:	PR#369

Issue:	Gsm4626: getGhoAmountForBuyAsset reports a slightly higher amount than necessary
Severity:	**Informational**
Violated property:	**getGhoAmountForBuyAsset_optimality**
Description:	When user wants to swap at least `min` GHO to assets, `getGhoAmountForBuyAsset(min)` should report the lowest amount of asset that will cost at least `min` GHO. In some cases the system recommends buying more assets (for more GHO) than what is necessary, i.e., the system encourages users to spend more GHO than needed. The error can be in the range of 10^-11 GHO.
Example:	Using `previewRedeem(shares) = RoundDown((shares*5)/3)`, `previewWithdraw(shares) = RoundUp((shares*3)/5)`. `price ratio = 10000000000000001`, `buyFee = 0`, `underlyingAssetDecimals = 22`, we have `getGhoAmountForBuyAsset(6) = (600000, 1, 1, 0)`, `getGhoAmountForBuyAsset(600000) = (1200000, 2, 2, 0)`
Mitigation/Fix:	Fixed in PR#369

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.

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

- The buy and sell fees have been assumed to be between 0 and 50%
- Unless specified otherwise, the decimals of gsm's underlying asset have been assumed to range from 5 to 27
- We assume that `PRICE_RATIO` between GHO and underlying is in the inclusive range $[10^{16}, 10^{20}]$
- In the case of the gsm's underlying asset being a 4626 share token:
  - The index used to convert shares into the 4626 underlying asset is 5/3. We use this under-approximation to allow modeling non-trivial rounding behaviour of an erc4626 token while reducing the technical complexity.
  - We assume that the 4626 token never collects fees
  - We assume that `previewWithdraw` and `previewMint` round up, and that `convertToShares` and `convertToAssets` round down
- While proving the properties, we assume that
  - `Gsm` 's GHO token is implemented by `GhoToken.sol`
  - `Gsm` 's fee strategy is implemented by `FixedFeeStrategy.sol`
  - `Gsm` 's price strategy is implemented by `FixedPriceStrategy.sol` and `Gsm4626` 's price strategy is implemented by `FixedPriceStrategy4626.sol`
- The implementation of openzeppelin's `node_modules/@openzeppelin/contracts/utils/math/Math.sol` was assumed to be correct
- We assume that `buyAsset` and `sellAsset` are not called with `Gsm` or its inheriting contracts as `msg.sender`

- We unroll loops. Violations that require executing a loop more than once will not be detected.
- We do not verify the cryptographic correctness of functions that involve calls to the `keccak256()` function.
- We do not verify function calls at `block.timestamp == 0`

Notations

✓ indicates the rule is formally verified on the latest reviewed commit.

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

Properties of Aave Gsm

Common Properties of Gsm and Gsm4626

Optimality of buy

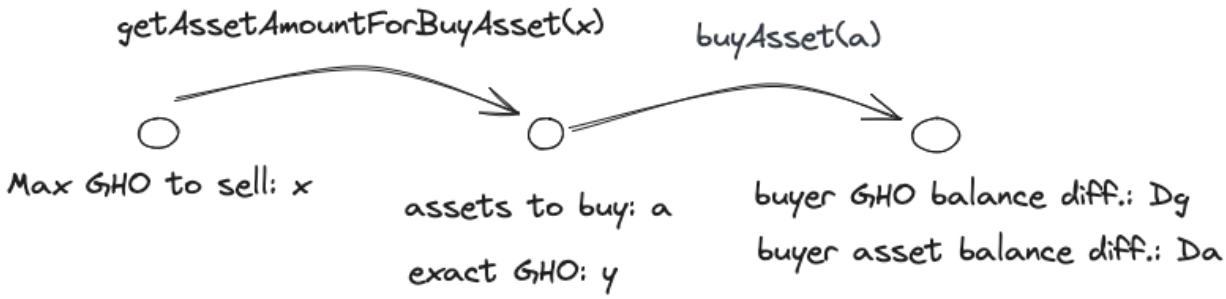
1. ❌ **getGhoAmountForBuyAsset_optimality**
   `getGhoAmountForBuyAsset(minAsset)` returns `finalAssetAmount` value that is as close as possible to user specified amount.
   - ✓ Verified after PR#369
2. ✓ **getAssetAmountForBuyAsset_optimality**
   `getAssetAmountForBuyAsset(maxGhoAmount)` returns `assetAmount` value that is as close as possible to user specified amount.

Optimality of sell

3. ✓ **getGhoAmountForSellAsset_optimality**
   `getGhoAmountForSellAsset(maxAssetAmount)` returns `finalAssetAmount` value that is as close as possible to user specified amount.
4. ❌ **getAssetAmountForSellAsset_optimality**
   `getAssetAmountForSellAsset(minGhoAmount)` returns `assetAmount` value that is as close as possible to user specified amount.
   - ✓ Verified after PR#369

Balances when buying

Fig. 1: Balances when buying

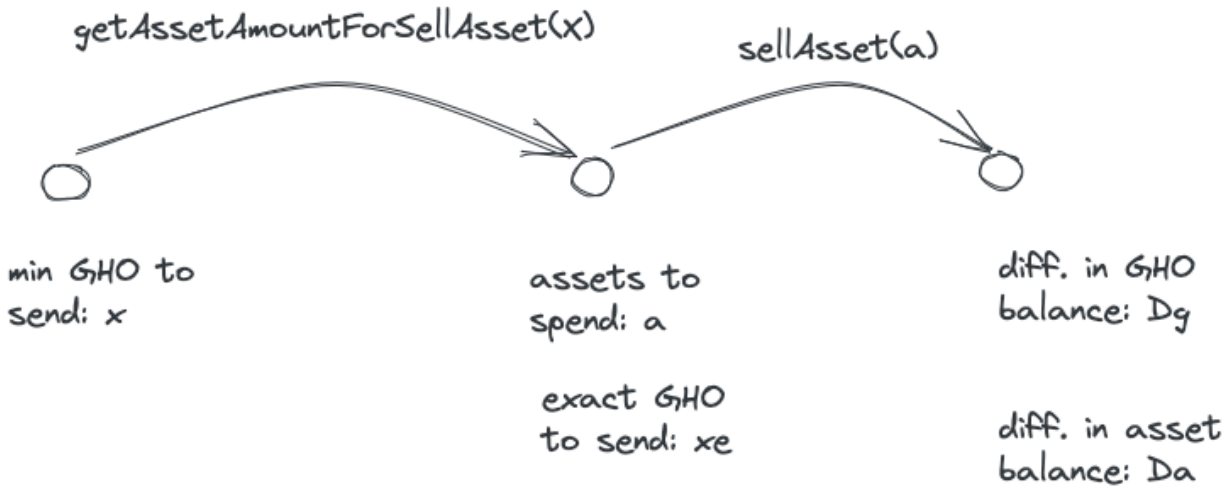


5. ❌ **R1_getAssetAmountForBuyAssetRV2** The exact amount of GHO `y` returned by `getAssetAmountForBuyAsset(x)` is less than or equal to `x` (Fig. 1)
   - ✓ Verified after PR#369
6. ❌ **R2_getAssetAmountForBuyAssetRV_vs_GhoBalance** The exact amount of GHO `y` returned by `getAssetAmountForBuyAsset(x)` matches the GHO amount `Dg` taken from user at `buyAsset(a)` (Fig. 1)
   - ✓ Verified after PR#369
7. ✓ **R3_buyAssetUpdatesAssetBuyerAssetBalanceLe** The increase in asset amount on user's account after `buyAsset(a)`, `Da`, is greater than or equal to `a` (Fig. 1)
8. ❌ **R4_sellGhoUpdatesAssetBuyerGhoBalanceGe** The amount of GHO `Dg` taken from user's account at `buyAsset(a)` is less than or equal to the value `x` passed to `getAssetAmountForBuyAsset(x)` (Fig. 1)
   - ✓ Verified after PR#369
9. ✓ **getGhoAmountForBuyAsset_correctness** `getGhoAmountForBuyAsset` never drops below the given bound

```
(finalAssets, _, _, _) = getGhoAmountForBuyAsset(minAssetAmount) ->
minAssetAmount <= finalAssets
```

Balances when selling

Fig. 2: Balances when selling

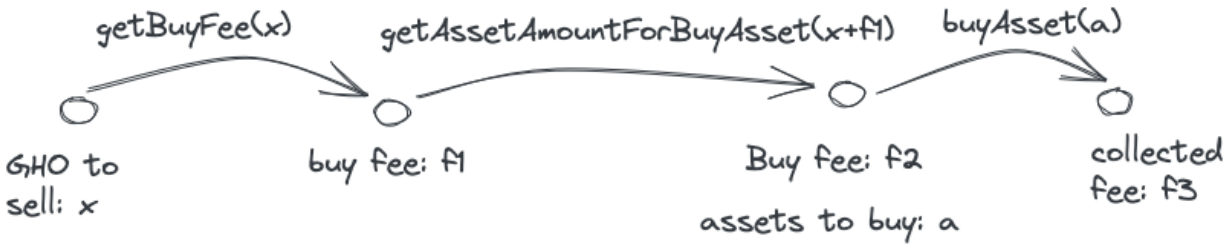


10. ✓ **R1_getAssetAmountForSellAsset_arg_vs_return** The exact amount `xe` of GHO returned by `getAssetAmountForSellAsset(x)` is greater than or equal to `x` (Fig. 2)
11. ✓ **R2_getAssetAmountForSellAsset_sellAsset_eq** The exact amount of GHO `xe` returned by `getAssetAmountForSellAsset(x)` is equal to the amount `Dg` obtained by the receiver after `sellAsset(a)` (Fig. 2)
12. ✓ **R3_sellAssetUpdatesAssetBalanceCorrectly** The asset amount `Da` taken from the user's account at `sellAsset(a)` is less than or equal to `a` (Fig. 2)
13. ✓ **R4_buyGhoUpdatesGhoBalanceCorrectly** The GHO amount `Dg` added to the user's account at `sellAsset(a)` is greater than or equal to the value `x` passed to `getAssetAmountForSellAsset(x)` (Fig. 2)
14. ✓ **getGhoAmountForSellAsset_correctness** `getGhoAmountForSellAsset` never exceeds the given bound

```
(finalAssets, _, _, _) = getGhoAmountForSellAsset(maxAssetAmount) ->
finalAssets <= maxAssetAmount
```

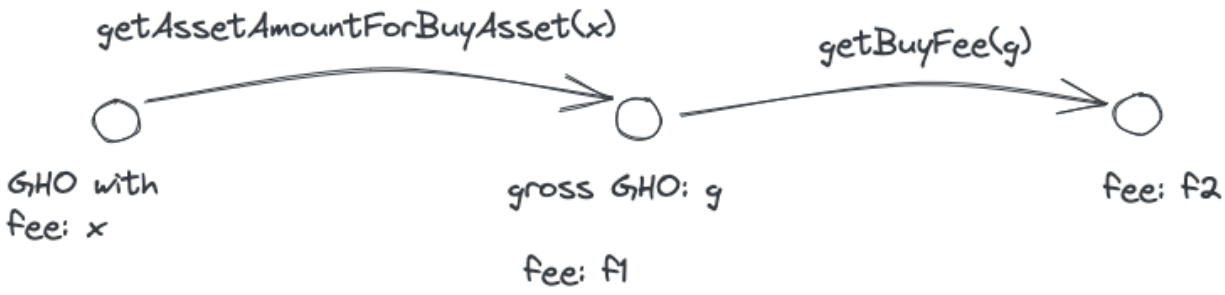
Fees when buying

Fig 3: Fees when buying



15. ✓ **R1_getBuyFeeGeGetAssetAmountForBuyAsset** The fee `f1` reported by `getBuyFee(x)` is greater than or equal to the fee `f2` reported by `getAssetAmountForBuyAsset(x+f1)` (see Fig. 3)
16. ✗ **R2_getAssetAmountForBuyAssetNeBuyAssetFee** The fee `f2` reported by `getAssetAmountForBuyAsset(x+f1)` is equal to the fee `f3` accrued by `buyAsset(a)` (see Fig. 3)
    - ✓ Verified after PR#369
17. ✗ **R4_estimatedBuyFeeGeActualBuyFee** The fee `f1` reported by `getBuyFee(x)` is greater than or equal to the fee `f3` accrued by `buyAsset(a)` (see Fig. 3)
    - ✓ Verified after PR#369

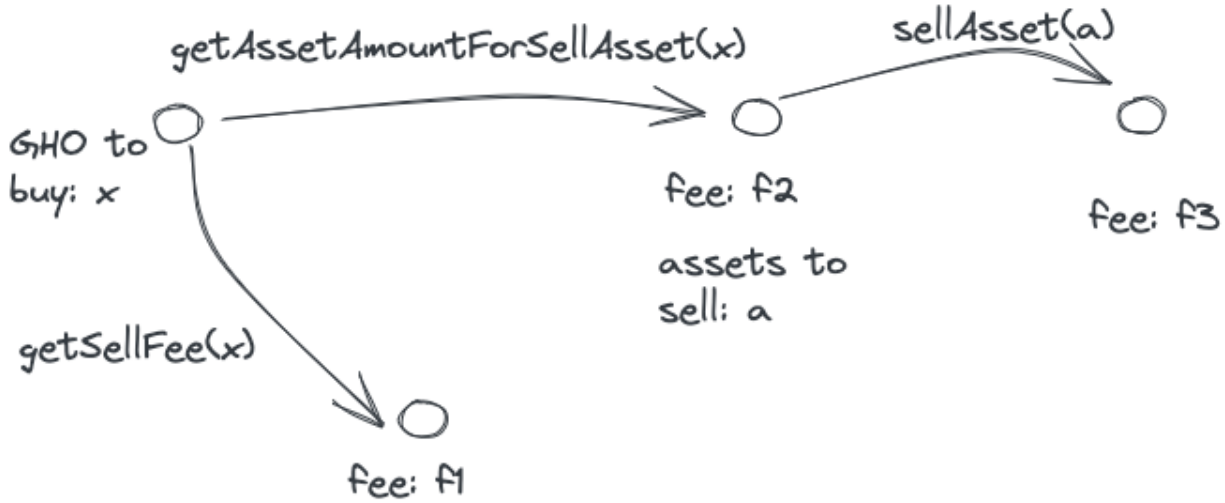
Fig. 4: Fees when buying 2



18. ✓ **R3_getAssetAmountForBuyAssetFeeEqGetBuyFee** The fee `f1` reported by `getAssetAmountForBuyAsset(x)` is equal to the fee `f2` reported by `getBuyFee(g)` where `g` is the gross GHO amount (see Fig. 4)
19. ✓ **NonZeroFeeCheckBuyAsset** If `buyFee percentage` $\gt$ `0` then amount of underlying received by user from `buyAsset` is less than `GHO spent / price ratio`

Fees when selling

Fig. 5: Fees when selling



20. ✓ **R1_getAssetAmountForSellAssetFeeGeGetSellFee** The fee `f2` reported by `getAssetAmountForSellAsset(x)` is greater than or equal to the fee `f1` reported by `getSellFee(x)` (Fig. 5)
21. ✓ **R2_getAssetAmountForSellAssetVsActualSellFee** The fee `f2` reported by `getAssetAmountForSellAsset(x)` is greater than or equal to the fee `f3` accrued by `sellAsset(a)` (Fig. 5)
22. ❌ **R3_estimatedSellFeeCanBeHigherThanActualSellFee** The fee `f1` reported by `getSellFee(x)` is less than or equal to the fee `f3` accrued by `sellAsset(a)` (Fig. 5)
    - ✓ Verified after PR#369
23. ✓ **R4_getSellFeeVsgetAssetAmountForSellAsset** The fee `f1` reported by `getSellFee(x)` is less than or equal to the fee `f2` reported by `getAssetAmountForSellAsset(x)` (Fig. 5)

24. ✓ **NonZeroFeeCheckSellAsset** If `sellFee percentage $\gt$ 0` then GHO received by user from `sellAsset` is less than `underlying amount * price ratio`

Frozen state

25. ✓ **cantBuyOrSellWhenFrozen, cantBuyOrSellWhenSeized** Buying/selling is not possible when the gsm is frozen and/or after it has been seized.
26. ✓ **rescuingGhoKeepsAccruedFees** Rescuing GHO never results in there being less GHO available (as an ERC-20 balance) in the gsm than `_accruedFees`.
27. ✓ **rescuingAssetKeepsAccruedFees** Rescuing the underlying asset never results in there being less of the underlying (as an ERC-20 balance) than `_currentExposure`.

Exposure

28. ✓ **sellingDoesntExceedExposureCap** It is not possible for `_currentExposure` of a gsm to exceed the `_exposureCap` as a result of a call to `sellAsset`.
29. ✓ **cantSellIfExposureTooHigh** If the `_currentExposure` exceeds the `_exposureCap`, `sellAsset` reverts until the `_currentExposure` is reduced below the `_exposureCap`.
30. ✓ **buyAssetDecreasesExposure** When calling `buyAsset` successfully (i.e., no revert), the `_currentExposure` always decreases.
31. ✓ **sellAssetIncreasesExposure** When calling `sellAsset` successfully (i.e., no revert), the `_currentExposure` always increases.
32. ✓ **giftingGhoDoesntAffectStorageSIMPLE** Gifting GHO does not affect storage.
33. ✓ **giftingUnderlyingDoesntAffectStorageSIMPLE** Gifting underlying asset does not affect storage.
34. ❌ **totalAssetsNotIncrease** For `price ratio == 1`, the total assets of a user do not increase, where total assets is defined as the sum of balances of the underlying asset and GHO converted to same units.
    - ✓ Verified after PR#369
35. ✓ **whoCanChangeExposureCap** Only `updateExposureCap`, `initialize` and `seize` methods can change `exposureCap`.

Fees

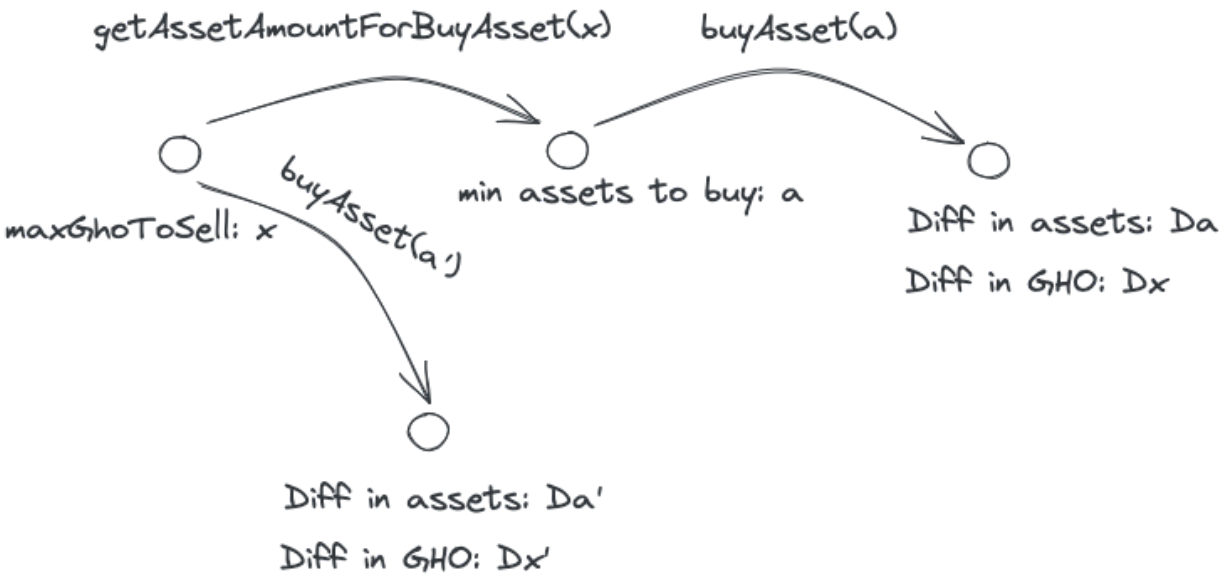
37. ❌ **collectedBuyFeeIsAtLeastAsRequired, collectedSellFeeIsAtLeastAsRequired** The fee actually collected (after rounding) is at least the required percentage.
    - Verified after PR#369

38. ✓ **whoCanChangeAccruedFees** `_accruedFees` never decrease, unless fees are being harvested by Treasury using `distributeFeesToTreasury`.
39. ✓ **accruedFeesLEGhoBalanceOfThis** `_accruedFees` $\le$ `ghotoken.balanceof(this)` is preserved by all methods.

Properties specific to non-4626 Gsm

Optimality of buy

Fig. 6: optimality of buy



40. ✓ **R2_optimalityOfBuyAsset_v2** non-4626: If user wants to sell at most `x` GHO, there is no better value `a'` to pass to `buyAsset` than the one given by `getAssetAmountForBuyAsset(x)`. In Fig. 6 above, there is no value `a'` such that `Da' >= Da` and `Dx' < x`, or `Da' > Da` and `Dx' <= x`
41. ✓ **getAssetAmountForBuyAsset_optimality**. non-4626: `getAssetAmountForBuyAsset` returns a value as close as possible to user specified amount. Let `(AssetAmount, _, _, _) = getAssetAmountForBuyAsset(maxGho)`. Then it is not possible to buy strictly more assets than `AssetAmount` while still paying less or equal to `maxGho`.
42. ✓ **R6_externalOptimalityOfBuyAsset** non-4626: The GHO sold by buying asset using values from `getAssetAmountForBuyAsset(maxGho)` is at least `maxGho - 2*oneAssetInGho + 1`. The lower bound is tight.

Optimality of sell

Fig. 7: Optimality of sell

43. ✓ * **R4_optimalityOfSellAsset_v2** non-4626: If user wants to buy at least `x` GHO, there is no better value `a'` to pass to `sellAsset` than the one given by `getAssetAmountForSellAsset(x)`. In Fig. 7 above, there is no value `a'` such that `Dx' <= x` and `Da > Da'` [1].
44. ✓ **getAssetAmountForSellAsset_optimality** non-4626: `getAssetAmountForSellAsset` returns a value as close as possible to user specified amount. Let `(AssetAmount, _, _, _) = getAssetAmountForSellAsset(minGho)`. Then it is not possible to sell strictly less assets than `AssetAmount` while still receiving more or equal to `minGho`.
45. ✓ **R5_externalOptimalityOfSellAsset** non-4626: The GHO received by selling asset using values from `getAssetAmountForSellAsset(minGho)` is upper bounded by `minGho + oneAssetinGho - 1`. The upper bound is tight.

Balances when buying

46. ✓ **monotonicityOfBuyAsset** non-4626: `buyAsset` is monotone (more asset bought <-> more GHO paid)

Balances when selling

47. ✓ **monotonicityOfSellAsset** non-4626: `sellAsset` is monotone (more asset sold <-> more GHO gained)

Selling and buying

48. ✓ **buySellInverse** non-4626: For price ratio == 1 and zero fees, `buyAsset` and `sellAsset` are inverse to each other.

Exposure

49. ✓ **enoughULtoBackGhoBuyAsset** non-4626: At every `buyAsset`, the insolvency of the contract will increase by at most `10^-18` GHO non-4626: gsm is always solvent `getAssetPriceInGho(_currentExposure) + 1` $\ge$ `ghoMinted`
50. ✓ **exposureBelowCap** non-4626: `currentExposure <= exposureCap` is preserved by all methods except `updateExposureCap` and `initialize`.
51. ✗ **systemBalanceStabilitySell** non-4626: The balance of the contract (difference between GHO minted and assets held by the contract converted to GHO value) can decrease by at most `10^-18` GHO after `sellAsset`.
    - ✓ Verified after PR#369
52. ✓ **whoCanChangeExposure** non-4626: Only `sellAsset`, and `sellAssetWithSig` can increase exposure. Only `buyAsset`, `seize` and `buyAssetWithSig` methods can decrease exposure.

OracleSwapFreezer

53. ✓ **freezeExecutable** `Freeze` action is executable if gsm is not seized, not frozen and price is lower than the freeze lower bound or higher than the freeze upper bound.
54. ✓ **unfreezeExecutable** `Unfreeze` action is executable if gsm is not seized, frozen, unfreezing is allowed and price is inside the unfreeze bounds.
55. ✓ **boundsAreContained** Unfreeze boundaries are contained in freeze boundaries

```
freezeLowerBound < unfreezeLowerBound
unfreezeUpperBound < freezeUpperBound
```

56. ✓ **freezeAndUnfreezeAreExclusive** There is no oracle price that allows both freeze and unfreeze.

FeeStrategy

57. ✓ **feeIsLowerThanGrossAmount** `getBuyFee(amount)` $\leq$ `amount`
58. ✓ **feeIsLowerThanGrossAmount** `getSellFee(amount)` $\leq$ `amount`
59. ✓ **GetSellFeeNeverReverts, GetBuyFeeNeverReverts** `getBuyFee` and `getSellFee` never revert
60. ✓ **getFeeIsMonotone** `getBuyFee` and `getSellFee` are monotone

```
(x1 <= x2) -> (getBuyFee(x1) <= getBuyFee(x2))
(x1 <= x2) -> (getSellFee(x1) <= getSellFee(x2))
```

61. ✓ **getGrossAmountFromTotalBought_isMonotoneInTotalAmount** `getGrossAmountFromTotalBought` is monotone

```
(x1 <= x2) ->
(getGrossAmountFromTotalBought(x1) <=
getGrossAmountFromTotalBought(x2))
```

62. ✓ **getGrossAmountFromTotalSold_isMonotoneInTotalAmount** `getGrossAmountFromTotalSold` is monotone

```
(x1 <= x2) ->
(getGrossAmountFromTotalSold(x1) <= getGrossAmountFromTotalSold(x2))
```

63. ✓ **getGrossAmountFromTotalSold_isCorrect** `getGrossAmountFromTotalSold` is inverse of `getSellFee`

```
y = getGrossAmountFromTotalSold(x) -> y - getSellFee(y) = x
```

64. ✓ **getGrossAmountFromTotalBought_isCorrect** `getGrossAmountFromTotalBought` is inverse of `getBuyFee`

```
y = getGrossAmountFromTotalBought(x) -> y + getBuyFee(y) = x + Delta
```

where `Delta` is either `-1, 0` or `1`

Properties specific to Gsm 4626

Optimality of sell and buy

65. ✓ **getAssetAmountForBuyAsset_optimality** 4626: `getAssetAmountForBuyAsset` returns a value as close as possible to user specified amount. Let `(AssetAmount, _, _, _) = getAssetAmountForBuyAsset(maxGho)`, Then it is not possible to buy strictly more assets than `AssetAmount` while still paying less or equal to `maxGho`.
66. ✓ **getAssetAmountForSellAsset_optimality** 4626: `getAssetAmountForSellAsset` returns a value as close as possible to user specified amount. Let `(AssetAmount, _, _, _) = getAssetAmountForSellAsset(minGho)`, Then it is not possible to sell strictly less assets than AssetAmount while still receiving more or equal to `minGho`.

Balances when buying

67. ✓ **R1_optimalityOfBuyAsset_v1** 4626: For asset values `a` given by `getAssetAmountForBuyAsset(x)`, the user can only get more assets by paying more GHO

Balances when selling

68. ✓ **R3_optimalityOfSellAsset_v1** 4626: For values `a` given by `getAssetAmountForSellAsset(x)`, the user can only gain more GHO by selling more assets

Changes in 4626 value

69. ✓ **backWithGhoDoesntCreateExcess, backWithUnderlyingDoesntCreateExcess** 4626: Sending GHO or the underlying asset via `backWith` does not result in a state where the asset immediately has yield to harvest.
70. ✓ **giftingGhoDoesntCreateExcessOrDearth** 4626: Gifting GHO doesn't create excess or dearth.
71. ✓ **giftingUnderlyingDoesntCreateExcessOrDearth** 4626: Gifting underlying asset doesn't create excess or dearth.

Exposure

72. ✓ **yieldNeverDecreasesBacking** 4626: Excess yield harvesting never results in previously-minted GHO becoming under-backed.
73. ✓ **exposureBellowCap** 4626: `currentExposure <= exposureCap` is preserved by all methods except `updateExposureCap`, `initialize` and `backWithUnderying`.
74. ✓ **whoCanChangeExposure** 4626: Only `sellAsset`, `sellAssetWithSig` and `backWithUnderlying` methods can increase exposure. Only `buyAsset`, `seize` and `buyAssetWithSig` methods can decrease exposure.

1. Not shown when the number of decimals in the underlying asset is 7, 10, 11, 12, 15, 16, and 17 ↩️