

# Formal Verification Report of Notional

## Summary

This document describes the specification and verification of Notional's smart contracts using the Certora Prover. The work was undertaken from June 28, 2021 to September 19, 2021.

The scope of our project was:

- LiquidityCurve
- BalanceHandler
- NoteERC20
- PortfolioHandler
- Incentives
- AccountActions
- BitmapAssets
- Settlement

The Certora Prover proved the implementation of the contracts correct with respect to the formal rules written by the Notional 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 Notional's smart contracts. All the rules are publically available in a public github: <https://github.com/notional-finance/contracts-v2/tree/certora-final-2>, <https://github.com/notional-finance/contracts-v2/tree/certora-liquidityCurve>.

# List of Main Issues Discovered

Severity: **High**

**Issue:** **Malicious users can settle asstes using settlement rates of other assets**

**Description:** When the function `SettlePortfolioAssets.settlePortfolio()` settles an asset in the portfolio that must be settled, it first computes its settlement rate. Due to an optimization, it only computes the settlement rate if `asset.maturity < blockTime`. However, it settles liquidity token assets if `asset.maturity <= blockTime`, meaning that if `asset.maturity` equals `blockTime` (`block.timestamp`), then because the assets in the portfolio are settled one by one in a loop, if this asset isn't the first one that is being settled in the loop, this asset will be settled with the settlement rate of the previous asset that was settled.  
`block.timestamp` can be manipulated by miners, and a malicious user (which is also a miner) can arrange the assets in its portfolio in such an order that allows him to earn significant funds on the expense of the system, by settleing assets with different settlement rates, at the date of their maturity.

**Mitigation/Fix:** Compute the settlement rate for an asset if `asset.maturity <= blockTime`, instead of only when `asset.maturity < blockTime`.

Severity: **Medium**

**Issue:** **Assets in a portfolio aren't always unique**

**Description:** If an asset is added to a portfolio using `PortfolioHandler.addAsset()` with `isNewHint=true`, but this asset already presents in the portfolio, then this asset will be inserted into the `portfolioState.newAssets` array.  
The effect of this is that the portfolio will end up having the same asset twice, instead of just one with its overall notional value. From now on, the second instance of that asset in the portfolio will be discarded.

**Mitigation/Fix:** Remove the argument `isNewHint` from `PortfolioHandler.addAsset()` and act like it is always false.

Severity: **Medium**

**Issue:** **Deleting an asset from a portfolio twice causes the loss of other assets**

Description: If the function PortfolioHandler.deleteAsset() tries to delete an asset that was already deleted (its storageState equals to AssetStorageState.Delete), then it will swap the deleted asset's storage slot (which is currently located after any storage slot of active assets) with a storage slot of an active asset. The result is that the swapped active asset will be stored after the slot of the deleted asset and portfolioState.storedAssetLength will be decreased by 1. Meaning that these two assets (the deleted asset and the swapped active asset) will not be loaded, when loading the portfolio from storage.

Mitigation/Fix: Add a require statement that prevents deletions of assets that were already deleted.

Severity: **Medium**

**Issue:** **SettlePortfolioAssets.settlePortfolio() also settles deleted assets**

Description: When the function SettlePortfolioAssets.settlePortfolio() settles the assets in the portfolio, it doesn't check whether the assets in the portfolio are active or they have been deleted, meaning that it also settles the deleted assets in the portfolio, if any. In addition, since every settlement also deletes the involved asset from the portfolio, settlement of a deleted asset will also trigger the previous issue (**Deleting an asset from a portfolio twice causes the loss of other assets**).

Mitigation/Fix: Settle only the active assets in the portfolio.

**Severity: Medium**

**Issue: Currency ids in an account context can be modified in AccountContextHandler.setActiveCurrency()**

**Description:** The 14 least significant bits of the flags argument in the function AccountContextHandler.setActiveCurrency() must be off because of these reasons:

- If isActive=false then the 14 least significant bits in flags can modify the 14-bits currencyId when it is found inside accountContext.activeCurrencies (it can set off bits in currencyId). Therefore, these 14 bits in flags must be set off.
- If isActive=true then the 14 least significant bits in flags can modify the 14-bits currencyId when it is found inside or inserted into accountContext.activeCurrencies (it can set on bits in currencyId). Therefore, these 14 bits in flags must be set off.

**Mitigation/Fix:** Add a requirement for these 14 bits to be off or set them off at the beginning of this function.

Severity: **Medium**

**Issue:**                    **The function `SettlePortfolioAssets.settlePortfolio()` can revert unexpectedly in valid transactions**

**Description:**        Recalling the previous issue with this function (**Malicious users can settle asstes using settlement rates of other assets**), if the asset that is being settled without computing its settlement rate (happens when `asset.maturity` equals `blockTime`) is the first asset in the loop that is being settled, then the settlement rate that will be used for its settlement will remain uninitialized, with the zero value. This will cause the settlement to failed due to a division by zero, when it calls the function `convertFromUnderlying()` and tries to devide by the uninitialized settlement rate.

**Mitigation/Fix:**     Compute the settlement rate for an asset if `asset.maturity <= blockTime`, instead of only when `asset.maturity < blockTime`.

**Severity: Medium**

**Issue: Authorized addresses can cause loss of asset tokens**

**Description:** Addresses that were authorized as global transfer operators using `GovernanceAction.updateGlobalTransferOperator()` are able to call `BatchAction.batchBalanceAction` and `BatchAction.batchBalanceAndTradeAction` with `account=Notional` (Notional's address) through `ERC1155Action._checkPostTransferEvent()`. With these functions, in the first action of the batch they can invoke `BalanceHandler.depositAssetToken()` with a token that has no transfer fee, and in the second action of the batch they can invoke `TokenHandler.redeem()`. This is what will happen in such a scenario:

- In the first action, when the function `BalanceHandler.depositAssetToken()` is invoked, `balanceState.netAssetTransferInternalPrecision` is increased according to the deposit amount. Then, in `BalanceHandler.finalize()`, the function `TokenHandler.transfer()` is invoked with the asset token and the value of `balanceState.netAssetTransferInternalPrecision` (converted to external precision), and transfer tokens from the account (itself) to itself, and no tokens are actually transferred.
- In the second action, in `BalanceHandler.finalize()`, the function `TokenHandler.redeem()` is invoked with the asset token and the amount we want to withdraw (it can be the entire balance of that account, we have just "deposited" tokens to it), redeeming the asset tokens that other users have deposited into the system, and then the function `TokenHandler.transfer()` is invoked with the underlying token, transferring the amount of underlying tokens that the system received from the redeem action, to itself. The system is now unaware that it holds this underlying tokens and they will be unreachable, and effectively lost.

**Mitigation/Fix:** Add a `require` statement in `TokenHandler.transfer()` that the account cannot be the contract itself (Notional's address), to prevent self transfers.

**Notional Response** We've added guards to prevent invalid addresses from accessing any of the trading or deposit actions in `ActionGuards.sol`

**Severity: Low**

**Issue: Missing validation of currencyId, maturity and assetType in PortfolioHandler.addAsset()**

Description: The function PortfolioHandler.addAsset() is missing require statements that validates the values of currencyId, maturity and assetType when a new asset is inserted into the portfolioState.newAssets array. The only validation is done when that portfolio is stored on storage, in PortfolioHandler.storeAssets(), during the internal call to \_encodeAssetToBytes().

Mitigation/Fix: Add require statements that validates these values.

**Severity: Low**

**Issue: Incorrect calculation of zero's MSB**

Description: The function Bitmap.getMSB() returns 0 for the input x=0, although the MSB of zero is not defined.

Mitigation/Fix: Revert if the input x is zero.

**Severity: Low**

**Issue: AccountContextHandler.setActiveCurrency() can insert an inactive currency id into an account context**

Description: If the arguments for AccountContextHandler.setActiveCurrency() are isActive=true and flags=0x0000 then the currencyId can be inserted into accountContext.activeCurrencies with no flags at all, meaning that it wasn't supposed to be inserted in the first place.

Mitigation/Fix: Add a require statement that prevents flags from being zero when isActive=true.

**Severity: Low**

**Issue: PortfolioHandler.buildPortfolioState() incorrectly implements an optimization for adding multiple new assets**

Description: The function PortfolioHandler.buildPortfolioState() should initialize state.newAssets to an array with length of newAssetsHint, as an optimization for a following addition of multiple new assets, but it doesn't do it if assetArrayLength=0.

Mitigation/Fix: Initialize state.newAssets also when assetArrayLength=0.

**Severity: Low**

**Issue: PortfolioHandler.addAsset() and PortfolioHandler.storeAssets() incorrectly use portfolioState.newAssets.length as the number of new assets.**

Description: The functions PortfolioHandler.addAsset() and PortfolioHandler.storeAssets() use portfolioState.newAssets.length as the number of new assets in the portfolio, although the portfolioState.newAssets array isn't necessarily full. The correct number of new assets is stored at portfolioState.lastNewAssetIndex. The value of portfolioState.newAssets.length can only be used as an upper bound for the number of new assets.

Mitigation/Fix: Use portfolioState.lastNewAssetIndex as the number of new assets in the portfolio, instead of portfolioState.newAssets.length.

**Severity: Low**

**Issue: Insufficient validation of currencyId in PortfolioHandler.\_encodeAssetToBytes()**

Description: The function PortfolioHandler.\_encodeAssetToBytes() only requires that currencyId will fit in uint16, but it also must not be greater than Constants.MAX\_CURRENCIES.

Mitigation/Fix: Change the current require statement to restrict currencyId to be less than or equal to Constants.MAX\_CURRENCIES.



**Severity: Recommendation**

**Issue: Unnecessary checks for a constant value in AccountContextHandler.setActiveCurrency()**

Description: The function AccountContextHandler.setActiveCurrency() checks the value of isActive 4 times on every loop iteration, although this value never changes.

Mitigation/Fix: Save gas by checking the value of isActive only once.

**Severity: Recommendation**

**Issue: Unnecessary array boundaries checks**

Description: When loading an array element more than once, there is no reason to check again that the index doesn't exceed the array limits.

Mitigation/Fix: Save gas by caching the array element in a local variable instead of loading it again.

**Severity: Recommendation**

**Issue: Computation of a value that isn't necessarily used in AccountContextHandler.isActiveInBalances()**

Description: The function AccountContextHandler.isActiveInBalances() computes isActive on every loop iteration, although it is not used in most iterations

Mitigation/Fix: Save gas by computing isActive only when it is necessary.

**Severity: Recommendation**

**Issue: Early exit optimization in AccountContextHandler.isActiveInBalances()**

Description: The function AccountContextHandler.isActiveInBalances() can return false in case it finds currencyId in the active currencies "array" and its ACTIVE\_IN\_BALANCES flag is off. There is no reason to continue iterating over the rest of the active currencies.

Mitigation/Fix: Save gas by returning false in this case, instead of continuing to iterate over the rest of the active currencies for no reason.

Severity: **Recommendation**

**Issue:** **Complicated require statement in AccountContextHandler.setActiveCurrency()**

Description: The complicated require statement after the while loop in AccountContextHandler.setActiveCurrency() that checks that the active currencies "array" contains less than 9 currencies, can be simplified to require(shifts < 9) because at this point, shifts equals the number of iterations that took place in the previous while loop that was iterating over this active currencies "array".

Mitigation/Fix: Save gas by simplifying this require statement to require(shifts < 9).

Severity: **Recommendation**

**Issue:** **Unnecessary castings in FloatingPoint56.unpackFrom56Bits()**

Description: The function FloatingPoint56.unpackFrom56Bits() contains two unnecessary castings from uint256 to uint256.

Mitigation/Fix: Remove these two unnecessary castings.

Severity: **Recommendation**

**Issue:** **Unnecessary casting in nTokenHandler.setNTokenCollateralParameters()**

Description: The function nTokenHandler.setNTokenCollateralParameters() contains an unnecessary casting from bytes32 to bytes32.

Mitigation/Fix: Remove this unnecessary casting.

Severity: **Recommendation**

**Issue:** **Unnecessary castings in nTokenHandler.setArrayLengthAndInitializedTime()**

Description: The function nTokenHandler.setArrayLengthAndInitializedTime() contains two unnecessary castings from uint256 to uint256.

Mitigation/Fix: Remove these two unnecessary castings.

Severity: **Recommendation**

**Issue:** **Trivial require statement in nTokenHandler.setArrayLengthAndInitializedTime()**

Description: The function nTokenHandler.setArrayLengthAndInitializedTime() requires lastInitializedTime to be greater than or equal to zero, even though it is a variable of type uint256.

Mitigation/Fix: Remove this trivial require statement.

Severity: **Recommendation**

**Issue:** **Trivial if statement in SettlePortfolioAssets.settlePortfolio()**

Description: The function SettlePortfolioAssets.settlePortfolio() contains an if statement that checks whether an asset is a fCash, followed by an “else if” statement that checks if that asset is a liquidity token, while the only two options available for that asset’s type are fCash and liquidity token.

Mitigation/Fix: Save gas by changing the “else if” statement to an “else” statement.

Severity: **Recommendation**

**Issue:** **Unnecessary require statement in DateTime.getTradedMarket()**

Description: The function DateTime.getTradedMarket() begins with the require statement require(index != 0). This requirement is unnecessary because even without it, if index=0, there will still be a revert at the end of the function.

Mitigation/Fix: Remove this unnecessary require statement.

**Severity: Recommendation**

**Issue:** Unnecessary checked arithmetic in `AssetHandler.getSettlementDate()`

**Description:** The `add(Constants.QUARTER)` operation in `AssetHandler.getSettlementDate()` performs an overflow check which is unnecessary because the function first subtracts `marketLength`, which is always greater than or equal to  $2 * \text{Constants.QUARTER}$  in this case, and only then adds `Constants.QUARTER` to it.

**Mitigation/Fix:** Save gas by replacing `add(Constants.QUARTER)` with `+ Constants.QUARTER`.

**Severity: Recommendation**

**Issue:** Trivial require statement in `BalanceHandler._setBalanceStorage()`

**Description:** The function `BalanceHandler._setBalanceStorage()` requires `lastClaimTime` to be greater than or equal to zero, even though it is a variable of type `uint256`.

**Mitigation/Fix:** Remove this trivial require statement.

## 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.

✗ 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  $\{p\} (C1 \sim C2) \{q\}$  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,  $C1 \sim_{op} C2$ , where  $op$  is a getter, indicating that  $C1$  and  $C2$  result in states with the same value for  $op$ .

## Verification of Market (Liquidity Curve)

Assumptions/Simplifications:

- Using a global context for market object.
- Simplified ABDK math

### Properties

1. Oracle rates are blended into the rate window given the rate oracle time window. ✓
2. Implied rates are correctly modified by the `executeTrade` function. ✓
3. Implied rates are not changed upon adding liquidity. ✓
4. Implied rates are not changed upon removing liquidity. ✓

## Verification of BalanceHandler

A library for handling deposits of underlying and asset tokens.

Assumptions/Simplifications:

- Using a global context for balance state object.
- ERC20 and Assets (cTokens) use a mock implementation.

### Properties

1. Integrity of deposit asset token (called from BalanceHandler) is preserved. ✓
2. Integrity of deposit asset token (called from AccountActions) is preserved. ✓
3. Integrity of deposit underlying token is preserved. ✓
4. Balance handler contract does not hold underlying in deposit. ✓

## Verification of NoteERC20

NoteERC20 is a standard ERC20 with delegations support, similar to COMP and Aave tokens.

### Properties

1. If A delegates to B and B delegates to C, then C's voting power is correctly updated in all possible cases: ✓
  - 1.1. A's previous delegatee and B's previous delegatee was C
  - 1.2. A's previous delegatee was not C and B's previous delegatee was C
  - 1.3. A's previous delegatee was C and B's previous delegatee was not C
  - 1.4. Neither A's nor B's previous delegatees were C
2. If A transfers tokens to B and B delegates to C, then C's voting power is correctly updated in all possible cases (as above). ✓
3. If A and B delegate to two different addresses, and A transfers tokens to B, voting power of A and B updates in the same way as their balances. ✓
4. Standard ERC20 properties - correctness of transfer functions. ✓

## Verification of nTokenAction

### Properties

1. The number of writes to account context is 0, or exactly 1 with exactly 1 read of the account context. ✗
  - Failure to adhere to this can lead to invalid account contexts

## Verification of Settlement

### Properties

1. Settlement rates are never reset. ✓
  - If settlement rates are reset they threaten the solvency of the system by allowing some accounts to settle for more or less cash than others.

## Verification of GovernanceAction

### Properties

1. Only the owner can call actions of GovernanceAction. ✓

## Verification of Bitmap

### Properties

1. Finding the most significant bit is correct ✓
2. Setting bits on and off are correct ✓


## Verification of DateTime

### Properties

1. Date math conversion between maturity and bitmap portfolio index (bitNum) are exact inverses of each other. ✓
2. All valid markets maturities can be converted to a market index (and vice versa) ✓

# Verification of Account Context

## Properties

1. Active currencies cannot be duplicated in the active currencies array -  (this is not passing due to performance limitations in handling quantified expressions)
2. A bitmap currency cannot be duplicated in the active currencies array ✓
3. Bitmap portfolios cannot have array assets ✓
4. Enabling a bitmap portfolio cannot leave behind stranded assets ✓