Intuition MultiVault v2

Date	September 2025
Auditors	George Kobakhidze, Heiko Fisch

1 Executive Summary

This report presents the results of our engagement with **Intuition** to review v2 of their **MultiVault contract and the accompanying bonding curves and bonding curve registry**.

The review was conducted from September 30 to October 10, 2025, by George Kobakhidze and Heiko Fisch.

The focus of this engagement was on the updated MultiVault system of contracts that make up the core of the Intuition Protocol's user experience. These contracts provide users with a way to stake TRUST to signal confidence in a "term", which – in its atomic form – may be an entity, event, or simply anything that could be a subject of a topic. Such terms could be combined with other terms to form a "triple", which constitutes a "subject", "predicate", and "object" relationship between the three terms. This triple combination would also be a "term", and could likewise be staked on. This allows Intuition users to express confidence in complex topics on-chain with actual monetary stakes. More details about the system can be found in the documentation.

This review is the second phase of the larger audit for the Intuition Protocol by Diligence. As such, it contains issues and references to code in the previous audit, specifically the `TrustBonding` system of contracts that provide token emissions and rewards streams. Indeed, one of the areas of review was the integration between the token emission system and the vault system in the protocol. Another area was the introduction of bonding curves to vaults that contain deposits tracking user signaling. Lastly, migration contracts were also reviewed to facilitate the new `MultiVault` deployment.

While most mechanisms present in the scope of this review seem simple in isolation, the complexity rises quickly upon integrating them all into one large system. Indeed, many of the significant issues presented in this report have to do with edge cases that appear out of sheer number of options available to `MultiVault` users. This is coupled with other integrating contracts such as `TrustBonding` and the bonding curves, which further expand the number of possible outcomes and edge cases with them.

The overall design of the `MultiVault` contracts is sound. The previous version of the protocol has been live for some time and has been "battle-tested" by real users. `TrustBonding` integration mechanisms are straightforward by themselves, though final rewards outcomes are hard to predict due to the number of factors involved.

The migration contract has no major issues but is heavily reliant on the deploying team to properly set up the data transfer process.

The newly introduced bonding curves are more rough around the edges than other parts of the protocol. The curve contracts in scope were inspired by and forked from other existing repositories, so choices such as style, libraries, logic checks, and other aspects are not consistent with the rest of the codebase. Indeed, it is important to note two areas of concern with the bonding curves. First, they are treated as mathematical black boxes by the `MultiVault` contracts. Curve integration is handled by an auxiliary curve registry contract, and the `MultiVault` contract mostly performs general invariant checks. Yet, it is difficult to generalize perfectly across all possible bonding curve implementations, so great care must be taken to test and cautiously integrate new bonding curves. Second, the curve contracts themselves should be implemented defensively, accommodate proper checks, and be aware of and enforce their own safe ranges to support error-free logic flow within the system.

The bonding curves reviewed in this audit do not appear to present any structural risks or design flaws. While the formulas are implemented correctly, the contracts exhibit issues similar to those found elsewhere in the system, particularly regarding edge cases and validation. Concerns like choice of calculation method, inaccurate limits, and disregard of rounding direction suggest that the bonding curves code is less production-ready compared to the rest of the codebase.

After delivery of the initial version of this report, the Intuition team has shared with us a set of PRs addressing findings from this report and an accompanying document that explains the fixes and/or provides comments on the individual issues. Some of these PRs – as well as the underlying issues – required further discussion between the Intuition and the auditing team. As a result, some PRs were amended with more commits, and sometimes a second PR was opened to further address the corresponding issue. Some findings in this report have also been amended and refined, following these discussions. As this process extended the fix review phase beyond what we had scheduled for it, fixes were reviewed on a best-effort basis. For each finding, we briefly describe in the "Resolution" box at its beginning how it has been dealt with.

2 Scope

This review focused on the following repository and code revision: 0xIntuition/intuition-contracts-v2, revision 2bb93d259c33d6d32843655ff936efc6bec421a9.

The detailed list of files in scope can be found in the Appendix.

3 System Overview

The items in scope of this review may be categorized into three primary sections: `MultiVault`, bonding curves, and the migration contract.

MultiVault:

The `MultiVault` system enables users to stake assets on arbitrary “terms” – which may represent entities, events, or any subject of interest within the Intuition protocol. Terms can also be combined into “triples” (subject, predicate, object), allowing users to express confidence in complex relationships on-chain by committing real value. This flexible staking mechanism underpins the protocol’s approach to decentralized signaling and topic curation. Further details on the term structure and user flows are available in the documentation.

User interactions with vaults – creating, depositing, withdrawing – generate fees that are routed according to vault and term type. For atomic terms, fees from all associated vaults are funneled into their pro-rata vault. Triple-based vaults similarly direct fees to their own pro-rata vault, while also allocating a portion to the underlying atom vaults that compose the triple. This layered fee flow ensures that both simple and composite topics are incentivized and rewarded appropriately.

A key addition in `MultiVault` v2 is the introduction of user utilization tracking, which integrates directly with the `TrustBonding` contracts. Utilization values, derived from user activity in the `MultiVault` system, are now an input for reward calculations in the broader Intuition protocol. This mechanism, while conceptually straightforward, required careful integration – particularly around epoch-based tracking – to avoid edge cases and gaming opportunities. Some of the issues identified in this review stem from the expanded set of user actions and entry points, which increase the complexity and potential for unexpected behaviors in utilization and reward outcomes.

Bonding curve registry and bonding curves:

As mentioned above, each term can have several associated vaults that users can stake into. Each term automatically receives a pro-rata vault upon creation, but other vaults that allow more complex economic games can be added later. Each vault’s behavior is governed by an underlying bonding curve, and bonding curves that can be used by the `MultiVault` will be added to the `BondingCurveRegistry` by its owner.

The `MultiVault` communicates with the bonding curves exclusively through this registry, which forwards all calls to the requested curve and returns the answer to the `MultiVault`. The curves themselves are essentially stateless; the `MultiVault` takes care of accounting (assets in the curve, total shares and share balances), and the deposited assets remain in the `MultiVault` too.

The following curves are currently implemented:

- `LinearCurve`: This curve defines a simple pro-rata vault. It accepts “donations”, i.e., assets added without a corresponding amount of shares minted will be distributed proportionally to the shareholders.
- `ProgressiveCurve`: This curve has a linear price function (and is otherwise often referred to as a “linear bonding curve”, but here, `LinearCurve` means the pro-rata vault described above). The slope is set in the constructor, so there can be different instances of the `ProgressiveCurve` with different slopes. “Donations” to the curve are not utilized and should therefore be avoided.
- `OffsetProgressiveCurve`: This is largely the same as `ProgressiveCurve` but also allows to define an “offset” in the constructor that shifts the curve to the left. An `OffsetProgressiveCurve` with offset zero behaves exactly like the `ProgressiveCurve` with the same slope.

MultiVaultMigrationMode

A special contract was introduced on top of the MultiVault contract that allows a privileged address to manually insert state data into the protocol. Administrators can insert atom data, triple data, and vault configurations without validation checks, requiring careful coordination during migration. This includes initializing vault states (total assets and shares) without performing balance verification – administrators must ensure proper funding before state insertion. Similarly, user balances are initialized through manual, surgical inserts that bypass normal accounting safeguards. The migration process places significant operational responsibility on the deployment team to maintain data consistency and prevent accounting mismatches between the old and new protocol states.

4 Security Specification

4.1 Actors & Trust model

The `MultiVault` is upgradeable, giving the entity with upgrade privileges complete control over all assets held by the `MultiVault`. Bonding curves to be used by the `MultiVault` can be added to the `BondingCurveRegistry` by its owner. While the registry and individual bonding curve implementations are not upgradeable, the registry to be used by the `MultiVault` can be changed by the administrator, so the same effect is achievable by replacing the bonding curve registry.

Apart from setting the registry, the `MultiVault` administrator can change several system parameters and configurations. During normal operations, admin activity is usually not required, and users are the only primary actors that create terms, deposit assets, and redeem shares from vaults.

5 Findings

Each issue has an assigned severity:

- Critical issues are directly exploitable security vulnerabilities that need to be fixed.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Issues without a severity are general recommendations or optional improvements. They are not related to security or correctness and may be addressed at the discretion of the maintainer.

5.1 Possible ID Collisions Between Atoms and Triples Major ✓ Fixed

Resolution

Fixed in PR 78 (reviewed commit 26ed09951a7fae2e47d65b30a8a8d442d7e38bff) by adding type salts before hashing.

Description

The IDs for atoms and triples are derived from their constituting data through hashing. More specifically, for an atom the `bytes` data that makes up the atom is hashed:

src/protocol/MultiVaultCore.sol:L157-L159

```
function calculateAtomId(bytes memory data) public pure returns (bytes32 id) {
    return keccak256(abi.encodePacked(data));
}
```

For a triple, the IDs of the three atoms that form the triple are concatenated and hashed:

src/protocol/MultiVaultCore.sol:L224-L234

```
function calculateTripleId(
    bytes32 subjectId,
    bytes32 predicateId,
    bytes32 objectId
)
    public
    pure
    returns (bytes32)
{
    return keccak256(abi.encodePacked(subjectId, predicateId, objectId));
}
```

Hence, for a given triple T, formed of atoms with the IDs `subjectId`, `predicateId`, and `objectId`, we can concatenate these three IDs, interpret the resulting `bytes` as raw data for an atom, and therefore, create an atom A with the exact same ID as T.

Essentially, this would block access to the triple, as the `getVaultType` function – which takes an ID and returns the type the ID belongs to – first checks if an *atom* with the given ID exists. Hence, an attacker could “block” arbitrary triples.

Recommendation

As part of the information to be hashed, a type distinguisher should be included. This could be an enum or a string or similar. For example, one might compute atom IDs as `keccak256(abi.encodePacked("ATOM", data))` and triple IDs as `keccak256(abi.encodePacked("TRIPLE", subjectId, predicateId, objectId))`.

5.2 Inaccurate Utilization Calculation and Rollover Issues Major ✓ Fixed

Resolution

Addressed in PR 88 (reviewed commit 1f9f4f966b16ef724e0348d9743899c8ba17e229) and PR 110 (reviewed commit b6314086c34f8465eb3467e2ceecde7ad0b3eb97). More specifically:

A. The wrong rewards calculation in `claimRewards` has been fixed by introducing a per-user history of the last three active epochs. As discussed in the finding, there are some `view` functions which take an epoch as input; these revert if the history is not sufficient to determine the correct result for the given epoch. Whether that causes any problems offchain has to be assessed by the client.

B. The Intuition team has decided to not change the smart contract code but to enforce system activity in every epoch. So the issue with the total utilization is acknowledged.

C. The rollover from epoch 0 has been fixed.

As the main problem of this finding has been resolved, we label is as “Fixed”, but it should be noted that part B requires further and continued action from the Intuition team, and this particular aspect should be considered “Acknowledged”.

Description

One of the major features of the Intuition Protocol is the notion of rewards based on user deposits that are fuelled by previous rewards. Specifically, the `TrustBonding` contract calls the `MultiVault` contract to query the user’s “utilization” via the function `getUserUtilizationForEpoch()`, as can be seen in this function:

src/protocol/emissions/TrustBonding.sol:L437-L454

```
function _getPersonalUtilizationRatio(address _account, uint256 _epoch) internal view returns (uint256) {
    if (_account == address(0)) {
        revert TrustBonding_ZeroAddress();
    }

    // If the epoch is in the future, return 0 and exit early
    if (_epoch > currentEpoch()) {
        return 0;
    }

    // In epochs 0 and 1, the utilization ratio is set to the maximum value (100%)
    if (_epoch < 2) {
        return BASIS_POINTS_DIVISOR;
    }

    // Fetch the personal utilization before and after the epoch
    int256 userUtilizationBefore = IMultiVault(multiVault).getUserUtilizationForEpoch(_account, _epoch - 1);
    int256 userUtilizationAfter = IMultiVault(multiVault).getUserUtilizationForEpoch(_account, _epoch);
```

src/protocol/MultiVault.sol:L214-L217

```
/// @inheritdoc IMultiVault
function getUserUtilizationForEpoch(address user, uint256 epoch) external view returns (int256) {
    return personalUtilization[user][epoch];
}
```

As mentioned above, user utilization is calculated from the user activity in the `MultiVault` contract. Namely, this utilization number goes up in an epoch if the user deposits more TRUST, and it goes down if the user withdraws TRUST. Additionally, this is recorded on a per-epoch basis. Indeed, in the function examples above, the `TrustBonding` contract calls `MultiVault`'s `getUserUtilizationForEpoch()` function twice - once for the `_epoch` value, and once for the `_epoch - 1` value. This is done to compare user activity between epochs and deduce how many rewards are appropriate for that user.

Since this is tracked on a per-epoch level, the user activity needs to transfer between epochs as well. To facilitate that process, a `_rollover()` function is introduced:

src/protocol/MultiVault.sol:L1405-L1407

```
if (personalUtilization[user][currentEpochLocal] == 0) {
    personalUtilization[user][currentEpochLocal] = personalUtilization[user][userLastEpoch];
}
```

As seen above, it quite simply allocates the utilization of the user's last active epoch to the current epoch.

However, the `TrustBonding` contract specifically calls `MultiVault` contract for the previous epoch's info, and the one before it. If the user skips performing any activity on any given epoch, they will have no utilization there, and the function `getUserUtilizationForEpoch()` for that epoch will simply return `0`. At the same time, once the user becomes active after this epoch has passed, their previous activity from a few epochs ago will roll over to the current epoch.

To put it simply, consider the case:

1. User deposits `1` TRUST, gaining utilization of 1 TRUST at `epoch=0`.
2. User skips any activity at `epoch=1`, achieving utilization of `0`.
3. User deposits - or may even withdraw - `1 WEI` worth of TRUST at `epoch=2`, rolling over their utilization from `epoch=0`, achieving the final utilization of `1+/-WEI` TRUST.
4. When claiming rewards for epoch=2 (in epoch 3), `TrustBonding` queries `MultiVault` for the user's utilization in epoch 2 and compares it to the stored utilization value for epoch 1. The resulting number will be `1+/-WEI-0=1+/-WEI`, effectively "crediting" themselves their previous activity for free.

This would allow the user to game the rewards system with minimal effort, ensuring they get maximum allocation from this part of the rewards equation.

There are two more aspects related to this issue:

1. Should there ever be an epoch with no activity at all, the same problem as described above for *personal* utilization occurs for *total* utilization tracking, as the mechanism to compute the delta is the same. In addition to that, the rollover is just always from the previous epoch, not from the last epoch *with activity*. Apparently, the assumption here is that there is never an epoch without any activity at all; if that is the case, the calculation of the total utilization delta is correct.
2. Currently, any user utilization at epoch 0 is not rolled over. It is treated as a special case and skipped. Specifically, in the case where a user was active in epoch 0, skipped a few epochs, and became active again later, not rolling over from 0 would provide incorrect delta utilization results.

Recommendation

A. In order to calculate user U's utilization delta for epoch e correctly, it must be taken into account that `personalUtilization[U][e]` contains U's utilization in epoch e *only if* U was active in epoch e; otherwise, the mapping entry remains zero. However, since a user's utilization doesn't change during epochs of inactivity, U's utilization in epoch e – assuming U was inactive in e – can be read from `personalUtilization[U][a]`, where a is the latest epoch not later than e in which U was active. In `claimRewards`, what is needed are the utilization values for current epoch – 1 and current epoch – 2. Hence, knowing – for every user – the last three epochs in which they were active is sufficient to correctly determine the utilization delta needed in `claimRewards`. It should be noted, though, that there are some `view` functions in `TrustBonding` (e.g., `getUserRewardsForEpoch`) that take an epoch as argument and would need a longer history of active epochs to work reliably for epochs older than current epoch – 1. While these are not used in state-changing functions, this could lead to UI or similar problems. Whether that is the case is beyond the scope of this review and has to be assessed separately by the client.

B. The same approach can be used to calculate the *total* utilization delta correctly. Again, it is sufficient to store the last three epochs in which there was any activity. While this is the solution we recommend, it is also possible to keep the current computation *and* enforce that there is some activity in every epoch. For example, the Intuition team could make sure to deposit in every epoch, or a bot could be set up to do that. With this approach, there is still some risk that depositing is forgotten nonetheless or that the bot dies, etc.

C. Fix the rollover from epoch 0.

5.3 Insufficient Onchain Checks for `MultiVaultMigrationMode` Migration Process Medium

Acknowledged

Resolution

Acknowledged and intended to be mostly checked offchain due to gas costs and contract size limitations. Added the `receive()` function to top off the contract balance to account for the migrated assets in PR 105 (reviewed commit ec4871abe004659e2c72a9ffe9ab45d1143fca14).

Description

The `MultiVaultMigrationMode` contract provides functionality for a privileged address with role `MIGRATOR_ROLE` to insert atoms, triples, and vault states. While the processes themselves do it correctly, there are insufficient checks on the provided data. This can be difficult to remediate after the fact. The potential problems could be the following:

1. Overwriting atom and triple data. During the processing of `batchSetAtomData()` and `batchSetTripleData()` functions, the provided data structures simply write directly into their associated provided termIDs. However, there could be a duplicate in the list, or an existing term's data structure prior to the migration, so the process could overwrite data.
2. Inaccurate asset event data. During `batchSetUserBalances()`, an event `Deposited` is emitted that contains what is intended to be the asset amount the user has deposited pre and post fees:

src/protocol/MultiVaultMigrationMode.sol:L189-L199

```
emit Deposited(
    address(this),
    params.user,
    params.termIds[i],
    params.bondingCurveId,
    assets,
    assets,
    params.userBalances[i],
    getShares(params.user, params.termIds[i], params.bondingCurveId),
    getVaultType(params.termIds[i])
);
```

However, this is calculated backwards from the user's recorded share amount at the vault's price point at the time of this migration, instead of the price point when the user deposited it. A user in question could have deposited midway through the vault's lifecycle when the fees only started to accumulate, or at the very end, after all other users, requiring that initial deposit to have a much larger asset amount. Moreover, one of the asset amounts in the event is meant to be the deposit after the fees, whereas in this case the event emits the same amount:

src/interfaces/IMultiVault.sol:L108-L118

```
event Deposited(
    address indexed sender,
    address indexed receiver,
    bytes32 indexed termId,
    uint256 curveId,
    uint256 assets,
    uint256 assetsAfterFees,
    uint256 shares,
    uint256 totalShares,
    VaultType vaultType
);
```

As a result, the event data is not completely accurate. This isn't used anywhere onchain and won't affect any user balances (as long as the migration data given was correct) but it could affect analytics and frontends integrating with the Intuition protocol. 3. Contract balance checks. After all the migration is done, it is important to ensure that all the balances from users add up to at most what is currently in their associated vaults, and all the vault balances sum up to less than or equal to the balance of the actual contract. Otherwise, the migration would allow for the contract to be underfunded.

Essentially, all of the above points are about migration validation which would help against providing incorrect data. Consider creating a separate event for the deposits during the migration to avoid confusing data for indexing services and analytics.

Recommendation

Consider adding validation checks to the migration data.

5.4 Some Calculations for General Vaults Assume Properties Only Valid for Pro-Rata Vaults Medium

✓ Fixed

Resolution

Fixed in PR 80 (reviewed commit 9d1cfb2d0cf989b2acee59a585127f8d303b2042) by calculating the required amount of assets to be taken from the deposits to mint `minShares` along the curve in the very beginning of each curve's state. This pair of `minShares` and associated asset amount is then first placed into the term's vault before any user deposits are processed.

Description

Whenever a new vault is created, `generalConfig.minShare` dead shares are minted. The amount of assets channeled into the vault for these dead shares is the exact same – also `generalConfig.minShare` :

src/protocol/MultiVault.sol:L1423-L1443

```
function _updateVaultOnCreation(
    address receiver,
    bytes32 termId,
    uint256 curveId,
    uint256 assets,
    uint256 shares,
    VaultType vaultType
)
    internal
    returns (uint256)
{
    uint256 minShares = generalConfig.minShare;
    VaultState storage vaultState = _vaults[termId][curveId];

    _setVaultTotals(
        termId,
        curveId,
        vaultState.totalAssets + assets + minShares,
        vaultState.totalShares + shares + minShares,
        vaultType
    );
```

In other words, there is always a 1:1 relationship between assets and shares assumed during vault creation. However, that is only correct for the creation of a *pro-rata* vault; in general, the amount of assets required to mint n shares is determined by the curve logic.

Moreover, when a new non-pro-rata vault is created via an initial deposit, the amount of shares that `MultiVault` mints for the user doesn't take the dead shares into account. These should, at least conceptually, be placed into the vault first, and only then should the shares for the user be minted. In the general case, these dead shares influence the price of the shares minted for the user, which is not the case for pro-rata vaults.

Recommendation

The amount of assets required to mint a certain amount of dead shares during vault creation should be determined by querying the curve. Moreover, it is not clear whether a one-size-fits-all amount of dead shares is appropriate for each and every curve, especially as more curves are added. It seems possible that a sensible amount of dead shares should be curve-dependent.

Finally, the amount of shares minted for an initial user deposit into a non-pro-rata vault has to take the dead shares correctly into account.

5.5 Shortcomings of the Calculation Method in `ProgressiveCurve` and `OffsetProgressiveCurve` Medium ✓ Fixed

Resolution

Fixed in PR 111 (reviewed commit 0506f0a295cd66de0a6651f244575525528c8813) as recommended.

Description

The way in which assets are converted into shares and vice versa in `ProgressiveCurve` and `OffsetProgressiveCurve` utilizes the prb-math library. More specifically, asset/share amounts are `convert` -ed into the `UD60x18` type, which means the integer n becomes the fixed-point number n.0, with 18 decimals. Calculations are then done in fixed-point arithmetic, and the end result is `convert` -ed back into an integer, which simply cuts off the decimal part.

For example, `previewDeposit` looks as follows in `ProgressiveCurve` :

src/protocol/curves/ProgressiveCurve.sol:L87-L103

```
function previewDeposit(
    uint256 assets,
    uint256, /*totalAssets*/
    uint256 totalShares
)
    external
    view
    override
    returns (uint256 shares)
{
    require(assets > 0, "Asset amount must be greater than zero");

    UD60x18 currentSupplyOfShares = convert(totalShares);

    return convert(
        currentSupplyOfShares.powu(2).add(convert(assets).div(HALF_SLOPE)).sqrt().sub(currentSupplyOfShares)
    );
```

This way of doing the calculations means that the number of decimals for the shares can, in principle, be freely chosen. For slope 2 in `ProgressiveCurve` and an asset like TRUST (or ETH) with 18 decimals, depositing initially 1 TRUST (i.e., 1e18 base units) results in 1e9 base-unit shares, so choosing 9 decimals for the shares would be a natural choice, as 1 TRUST would then be converted into 1 share with 9 decimals.

However, for several reasons, it would be desirable to have shares with 18 decimals: First of all, 18 decimals are more or less standard and often expected. Second, the shares for `LinearCurve` have 18 decimals too. (At least, that's the natural choice.) So having a different number of decimals for different curves adds complications or is, at the very least, inconvenient and error-prone. As mentioned above, the number of decimals can, in principle, be chosen, but choosing 18 decimals in the example above (`ProgressiveCurve` with slope 2) means 1 TRUST would be converted into 1e-9 = 0.000000001 shares – an inconveniently small number.

In order to avoid this problem, a very small slope has been chosen: 2e-18. The effect of this is that base unit share numbers will be multiplied by 1e9 compared to slope 2, so we get the same share numbers with 18 decimals which we'd get for slope 2 and 9 decimals. In a way, this solves the problem: For slope 2e-18, depositing 1 TRUST now results in 1 share with 18 decimals, so we have "natural" share numbers and 18 decimals – but a very small slope. In fact, as the slope has only 18 decimals, this is very close to the smallest slope representable. Intuitively, though, the curve is quite steep: The first share costs 1 TRUST, the next one already costs 3 TRUST, the next one 5, etc. Having the possibility to define curves with a more moderate price increase would probably be desirable.

Recommendation

In order to achieve this and still have shares with 18 decimals and "natural" numbers, we recommend replacing

- every `convert` for the direction `uint256` to `UD60x18` with `wrap`; and
- every `convert` for the direction `UD60x18` to `uint256` with `unwrap`.

The current slope 2e-18 can then be replaced with 2 to get the same curve behavior.

This wrap/unwrap logic was already used in a previous version of the code. In an engagement earlier this year, when the curves were out of scope, we gave them a cursory look upon the client's request and – mistakenly – flagged this implementation as wrong. With the more thorough look we could afford this time, it became apparant to us that wrap/unwrap is indeed the better choice.

The effect of this change is that the actual calculations now don't happen on *base unit* assets and *base unit* shares anymore, but on 18-decimals TRUST numbers and shares with 18 decimals as well. In other words, this change hard-codes 18 decimals into the shares and also assumes that the asset has 18 decimals. Now, with slope 2, depositing 1 TRUST will again result in 1 share, but we have room for smaller slopes.

Remark

The curve-related findings are not independent of each other. We think it makes sense to deal with them in the order in which they are presented in this report, but they should still be approached with an awareness for tie-ins. This issue is the most basic one and should be addressed first.

5.6 Rounding Direction in Preview Functions Medium ✓ Fixed

Resolution

Fixed in PR 93 (reviewed commit c9d204340c3d63b50c01487128dc4f5d9e5a70a0) and PR 111 (reviewed commit 0506f0a295cd66de0a6651f244575525528c8813) as recommended.

Description

As a general rule for vaults, rounding should happen in the direction that benefits the protocol (as opposed to the user). This means that functions that compute an "amount out" should round down, and functions that calculate an "amount in" should round up.

A. Currently, the only two preview functions that `MultiVault` actually uses are `previewDeposit` and `previewRedeem`; these two are "amount out" functions and should, therefore, round down. That is the case in `LinearCurve`. In `ProgressiveCurve` and `OffsetProgressiveCurve`, it is the case *essentially* but with one caveat: `previewDeposit` divides by `HALF_SLOPE`, which is itself a number that got rounded down by halving:

src/protocol/curves/ProgressiveCurve.sol:L69

```
HALF_SLOPE = UD60x18.wrap(slope18 / 2);
```

As dividing by a rounded down number might actually round up, we'd want to avoid that. However, this is not a problem if `slope18` is an even number. In that case, dividing by 2 is exact..

B. The other two preview functions – `previewWithdraw` and `previewMint` – should round up, as they are "amount in" functions. That is not the case, though. Since these functions are currently not used, this is not a real problem at the time. However, unlike `MultiVault`, the curve contracts are not upgradeable, so this can't be (easily) fixed later; we discuss this in more detail in issue 5.14.

Recommendation

First of all, issue 5.5 should be addressed and the changes implemented. After that, we recommend the following:

A. The constructors in `ProgressiveCurve` and `OffsetProgressiveCurve` should enforce that `slope18` is an even number. In that case, dividing by 2 is exact and we don't have to worry about the rounding direction of this term. And this restriction is hardly a limitation in practice (once issue 5.5 has been fixed).

B. In `LinearCurve`, `ProgressiveCurve`, and `OffsetProgressiveCurve`, ensure that `previewDeposit` and `previewRedeem` round down. `previewWithdraw` and `previewMint`, on the other hand, should round up. This can be more subtle than it might a seem at first, as an intermediate expression has to be rounded in the opposite direction of the (current) target direction if it is subtracted or divided by. For compatibility with ERC-4626, it seems best to have `convertToAssets` and `convertToShares` round down, but these functions are currently not used either.

5.7 Issues Related to Limits in the Curve Contracts Medium ✓ Fixed

Resolution

Fixed in PR 103 (reviewed commit 6f456994e96229bd0d03fb507f555e02871fe660) and PR 111 (reviewed commit 0506f0a295cd66de0a6651f244575525528c8813) as recommended.

Description

The curve contracts have to provide functions `maxAssets` and `maxShares` that are supposed to return the maximum number of assets (shares, resp.) that the curve can "hold." We assume the idea here is to ensure that the `totalAssets` and `totalShares` "in" the curve never reach values at which a (possibly intermediate) computation overflows and it is not possible anymore to convert shares back into assets via `previewRedeem` or `previewWithdraw`. The other direction, i.e., reverting when assets are deposited (or shares minted) is less problematic, as this just means we can't get these assets *in*.

The functions that are/should be called in `MultiVault` to retrieve these values are `getCurveMaxAssets` and `getCurveMaxShares` on the curve registry, which then translates this into a `maxAssets` / `maxShares` call to the corresponding curve contract:

src/protocol/curves/BondingCurveRegistry.sol:L241-L255

```
/// @notice Get the maximum number of shares a curve can handle.  Curves compute this ceiling based on their
/// constructor arguments, to avoid overflow.
/// @param id Curve ID to query
/// @return maxShares The maximum number of shares
function getCurveMaxShares(uint256 id) external view returns (uint256 maxShares) {
    return IBaseCurve(curveAddresses[id]).maxShares();
}

/// @notice Get the maximum number of assets a curve can handle.  Curves compute this ceiling based on their
/// constructor arguments, to avoid overflow.
/// @param id Curve ID to query
/// @return maxAssets The maximum number of assets
function getCurveMaxAssets(uint256 id) external view returns (uint256 maxAssets) {
    return IBaseCurve(curveAddresses[id]).maxAssets();
}
```

While each curve contract is free to implement these functions in its own way, the currently existing curves define constants `MAX_ASSETS` and `MAX_SHARES`, which the `maxAssets` and `maxShares` functions simply return, respectively.

There are currently several problematic aspects with this mechanism:

A. `getCurveMaxShares` is never queried at all in `MultiVault`. It is conceivable that the intention is that using `getCurveMaxAssets` *alone* should be sufficient to prevent overflows in `previewRedeem` and `previewWithdraw`, but that perspective raises the question why `getCurveMaxShares` should be provided in the first place. We also think that it is not a bad idea to err on the side of caution and check both limits explicitly.

B. `getCurveMaxAssets` is only called once, in `_validateMinShares`, which is called as part of the validation when there is a deposit into that curve.

src/protocol/MultiVault.sol:L1538-L1539

```
function _validateMinShares(bytes32 _termId, uint256 _curveId, uint256 _assets, uint256 _minShares) internal view {
    uint256 maxAssets = IBondingCurveRegistry(bondingCurveConfig.registry).getCurveMaxAssets(_curveId);
```

However, pro-rata vaults receive new assets "on the side" (i.e., without shares being minted), as part of deposits into other curves, and these additions are not checked to stay within the limit that the linear curve defines.

C. `LinearCurve` defines the maximum amount of assets/shares the curve can hold both as `type(uint256).max`:

src/protocol/curves/LinearCurve.sol:L24-L29

```
/// @dev Maximum number of shares that can be handled by the curve.
uint256 public constant MAX_SHARES = type(uint256).max;

/// @dev Maximum number of assets that can be handled by the curve.
uint256 public constant MAX_ASSETS = type(uint256).max;
```

If the curve really held that many assets, the following code would revert if someone tried to redeem with `shares > 1`:

src/protocol/curves/LinearCurve.sol:L120

```
assets = supply == 0 ? shares : shares.mulDiv(totalAssets, supply);
```

An alternative to changing the limits is the replacement of the two occurrences of `mulDiv` in `LinearCurve` with `fullMulDiv`; then the current limits can be kept. Finally, it should be said that these considerations are more of a theoretical nature: According to the client, "TRUST supply is capped (1B with single-digit annual inflation)", so the numbers that can occur in practice are much smaller.

D. In `ProgressiveCurve`, `MAX_SHARES` is too generous; if there were that many `totalShares`, trying to redeem any amount of shares would revert. As mentioned in (A), this value is never queried, though, so changing it would currently not have any effect.

Since the total amount of assets in the progressive curve doesn't have to be considered in the `previewRedeem` or `previewWithdraw` calculation, it can't (directly) lead to an overflow in that function and, as a consequence, `MAX_ASSETS` could even be set to `type(uint256).max` without causing any harm in that regard. Of course, this assumes (1) that we do check `MAX_SHARES` and don't rely on a `MAX_ASSETS` check alone to indirectly limit the number of shares that can be minted. (This was briefly discussed as a possibility in A.) Moreover, (2) it should be noted that `ProgressiveCurve` currently doesn't allow the addition of assets without minting a corresponding amount of shares. If that is ever changed, the total amount of assets will have to be used in the calculations and, therefore, could lead to an overflow. That all said, we think a sensible approach here is to set `MAX_ASSETS` to a meaningful value, i.e., the amount of assets that corresponds to `MAX_SHARES` shares.

E. The situation in `OffsetProgressiveCurve` is analogous to `ProgressiveCurve` (D).

Recommendation

First of all, issue 5.5 should be addressed and the changes implemented. After that, we recommend the following:

A and B. In function `_setVaultTotals`, check that the new `totalAssets` and `totalShares` don't exceed the limits defined by the curve. Also, add to `_validateMinShares` a check that the new total amount of shares doesn't exceed the limit defined by the curve.

C. In `LinearCurve`, replace the two occurrences of `mulDiv` with `fullMulDiv`.

D and E. In each of these two curves, `MAX_SHARES` should be set to a (sufficiently large) number that avoids an overflow in `previewRedeem` and `previewWithdraw`. `MAX_ASSETS` should be set to the amount of assets that corresponds to `MAX_SHARES` shares.

5.8 Lack of Sanity Checks in Bonding Curve Implementations Medium ✓ Fixed

Resolution

Fixed in PR 102 (reviewed commit ec7daf3aa82132f4f95d630e5718a3ca636a2306) and PR 111 (reviewed commit 0506f0a295cd66de0a6651f244575525528c8813) as recommended.

Description

The backbone of the curve contracts are the preview functions: `previewDeposit`, `previewRedeem`, `previewWithdraw`, and `previewMint`, where the last two are currently not used in the MultiVault but might be at a later point. The only sanity check in these four functions is in `ProgressiveCurve`'s and `OffsetProgressiveCurve`'s `previewDeposit` function, which checks that the amount to deposit is non-zero – and that particular check might be considered overzealous, as there is, technically, no problem with depositing zero assets. And while it is *pointless* to deposit zero assets, reverting in this case might be fairly opinionated for a curve contract. Moreover, the `LinearCurve` doesn't revert if someone tries to deposit zero assets, and it would make sense for all curves to behave identically in this scenario.

On the other hand, the curves accept inputs and return values that would exceed the amount of shares/assets that the curve can hold. And the `LinearCurve` doesn't complain if someone tries to withdraw/redeem more assets/shares than exist in total. Of course, this should be caught elsewhere, but as an overall strategy, we think it makes sense to have multiple layers of protection and have functions reject arguments or argument combinations that don't make sense in the first place.

Sanity checks that the curves could – and probably even should – perform:

1. `totalAssets <= maxAssets()` and `totalShares <= maxShares()` in `previewDeposit`, `previewRedeem`, `previewWithdraw`, `previewMint` (and `convertToAssets`, `convertToShares`). (Some of the checks below subsume this one, e.g., if `assets + totalAssets <= maxAssets()` is true, then `totalAssets <= maxAssets()` is true as well. Nevertheless, the simplest approach might still be to include this as a general sanity check at the beginning of each function.)
2. `assets <= totalAssets` in `previewWithdraw`
3. `shares <= totalShares` in `previewRedeem` (and `convertToAssets`)
4. `assets + totalAssets <= maxAssets()` in `previewDeposit` (and `convertToShares`)
5. `shares + totalShares <= maxShares()` in `previewMint`
6. `return value + totalAssets <= maxAssets()` in `previewMint`
7. `return value + totalShares <= maxShares()` in `previewDeposit` (and `convertToShares`)

Recommendation

1. In `ProgressiveCurve` and `OffsetProgressiveCurve`, consider removing the checks for a non-zero `assets` amount in `previewDeposit` and `convertToShares` to make the curve implementation less opinionated. In situations where reverting is desired for `asset` amount zero, this should be handled by the caller, instead of the curve itself.
2. For all curves, add the checks outlined above. Ideally, this is done in a way that avoids code duplication and ensures the checks are consistently applied in all curves.

5.9 `totalTermsCreated` Counter Not Incremented for Counter Triples Minor ✓ Fixed

Resolution

Fixed in PR 101 (reviewed commit 8ab08ed117ecb3aa774e21e828254ca3c083f196) as recommended.

Description

The `totalTermsCreated` counter is only incremented once during the triple vault creation flow. However, counter triples are created at the same time, so `totalTermsCreated` should be increased by two, not one.

src/protocol/MultiVault.sol:L635

```
++totalTermsCreated;
```

Recommendation

Increase `totalTermsCreated` by two to accurately track the term numbers.

5.10 Missing Events in Configuration Setters Minor ✓ Fixed

Resolution

Fixed in PR 100 (reviewed commit d09af18a6b72df35d4e36dd8de86f045a726248f) by adding events to the setter functions. The NatSpec was fixed to inherit from the interface correctly.

Description

In `MultiVault`, there are several setter functions to change configuration parameters:

src/protocol/MultiVault.sol:L954-L985

```
/// @notice returns the general configuration struct
function setGeneralConfig(GeneralConfig memory _generalConfig) external onlyRole(DEFAULT_ADMIN_ROLE) {
    _setGeneralConfig(_generalConfig);
}

/// @notice returns the atom configuration struct
function setAtomConfig(AtomConfig memory _atomConfig) external onlyRole(DEFAULT_ADMIN_ROLE) {
    atomConfig = _atomConfig;
}

/// @notice returns the triple configuration struct
function setTripleConfig(TripleConfig memory _tripleConfig) external onlyRole(DEFAULT_ADMIN_ROLE) {
    tripleConfig = _tripleConfig;
}

/// @notice returns the vault fees struct
function setVaultFees(VaultFees memory _vaultFees) external onlyRole(DEFAULT_ADMIN_ROLE) {
    vaultFees = _vaultFees;
}

/// @notice returns the bonding curve configuration struct
function setBondingCurveConfig(BondingCurveConfig memory _bondingCurveConfig)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    bondingCurveConfig = _bondingCurveConfig;
}

/// @notice returns the wallet configuration struct
function setWalletConfig(WalletConfig memory _walletConfig) external onlyRole(DEFAULT_ADMIN_ROLE) {
    walletConfig = _walletConfig;
}
```

These functions should all emit an event to inform about this important configuration change.

Recommendation

Emit an appropriate event in all these functions. (Also, the comments for these functions are wrong.)

5.11 `previewDeposit` and Other Similar `view` Functions Do Not Check for Term Existence Minor ✓ Fixed

Resolution

Fixed in PR 99 (reviewed commit 4bea39fe7c086339e204f5b20ae118aeed808635) by introducing `_isTermCreated` checks to the appropriate view functions.

Description

The view functions (`previewDeposit` , `previewRedeem` , `convertToShares` , `convertToAssets`) lack proper validation checks that are present in their corresponding state-changing counterparts, creating API inconsistencies and potential integration issues. While these are view functions that don't affect state, the validation gaps create unreliable user interfaces and integration points.

Namely, these functions do not check for the existence of the provided `termID` . In the case of `convert*` functions, this simply ends up passing `0` values for the vault states to the bonding curves, which could be considered valid, depending on the curve. However, in the case of `preview*` functions, this forces the code to assume that the provided `termID` is actually a triple. This is because of the `isAtom()` check:

src/protocol/MultiVault.sol:L300-L311

```
function previewDeposit(
    bytes32 termId,
    uint256 curveId,
    uint256 assets
)
    public
    view
    returns (uint256 shares, uint256 assetsAfterFees)
{
    bool _isAtom = isAtom(termId);
    return _calculateDeposit(termId, curveId, assets, _isAtom);
}
```

src/protocol/MultiVaultCore.sol:L175-L177

```
function isAtom(bytes32 atomId) public view returns (bool) {
    return _atoms[atomId].length != 0;
}
```

Which in the following section treats the term as a triple:

src/protocol/MultiVault.sol:L1027-L1042

```
function _calculateDeposit(
    bytes32 termId,
    uint256 curveId,
    uint256 assets,
    bool isAtomVault
)
    internal
    view
    returns (uint256 shares, uint256 assetsAfterFees)
{
    if (isAtomVault) {
        return _calculateAtomDeposit(termId, curveId, assets);
    } else {
        return _calculateTripleDeposit(termId, curveId, assets);
    }
}
```

Thereby providing triple deposit data for a non-existing term.

As mentioned above, this is coupled with necessary checks in the state-changing parts of the code, so currently no incorrect logic is present. However, it would be better for the associated `view` functions to also perform relevant existence checks so that integrating systems and frontends do not report false data.

Recommendation

Consider implementing additional checks to the view functions `previewDeposit` , `previewRedeem` , `convertToShares` and `convertToAssets` such as `termId` existence checks.

5.12 Missing Sanity Checks for Curve ID Validity Minor ✓ Fixed

Resolution

Fixed in PR 98 (reviewed commit 35fc91fe9f89b1ad655b7e81489f40046f700a59) by adding a `onlyValidCurveId` modifier to data-retrieving functions of the `BondingCurveRegistry` contract.

Description

For several external and public functions, the caller can specify a `curveId` . This ID is not checked to be valid, neither in the top-level function, nor later. In the end, if an invalid ID is provided, `BondingCurveRegistry` tries to forward a call to the zero address – which is the address associated with an invalid ID – and we'll end up with a low-level compiler-inserted revert. It would be cleaner to catch such situations with a proper user-defined error.

Recommendation

Before trying to make a call, `BondingCurveRegistry` should always check if the given curve ID is valid and revert with a proper user-defined error if not. We recommend a modifier for that. Moreover, it's not uncommon to sanity-check user inputs in the top-level function. If that is desired, a function that returns whether a given curve ID is valid could be added to `BondingCurveRegistry` and then queried in `MultiVault` functions like `deposit`, `redeem`, etc.

5.13 `previewAtomCreate` and `previewTripleCreate` Should Not Have a `curveId` Parameter

Minor ✓ Fixed

Resolution

Fixed in PR 97 (reviewed commit aeca8e34bbaab7d4bc50405594a610ed8c175c2f) by removing the `curveId` parameter from `previewAtomCreate` and `previewTripleCreate` functions.

Description

The view functions `previewAtomCreate` and `previewTripleCreate` both have a `curveId` parameter, but actual creation of atoms and triples always uses the default curve – which, for the current state of the codebase, has to be `LinearCurve`.

Hence, calling these functions with a curve ID that is different from the ID of the default curve, returns a meaningless number.

Recommendation

As the curve used in atom or triple creation can't be chosen, `previewAtomCreate` and `previewTripleCreate` should not take a `curveId` parameter in the first place and use `bondingCurveConfig.defaultCurveId` instead to pass as `curveId` to `_calculateAtomCreate` / `_calculateTripleCreate`.

5.14 Consider Making Curve Contracts and Curve Registry Upgradeable

Minor ✓ Fixed

Resolution

Fixed in PR 95 (reviewed commit f80118bd9a462d615c89b146f974cf8e99e78858) by making the `BondingCurveRegistry`, `BaseCurve`, and individual curve implementation contracts upgradeable.

Description

The curve contracts implement – and *have to* implement, as this is required by the interface – several functions that are not used by `MultiVault`. In particular, `MultiVault` doesn't use `previewWithdraw`, `previewMint`, `convertToAssets`, and `convertToShares`, while it does use `previewDeposit` and `previewRedeem`.

According to the client, this is to future-proof the curve contracts, as `MultiVault` might be upgraded at some point to utilize these functions. Notably, `MultiVault` is upgradeable, while the curve contracts are not. (However, the curve registry, which facilitates communication between `MultiVault` and registered curve contracts, can be changed in `MultiVault`; that would still allow a (cumbersome) "upgrade" of curve contracts.)

We consider it questionable whether the goal of really future-proof curve contracts is attainable, especially at such an early point. It is conceivable that, as the `MultiVault` code advances, more curve-dependent information will be needed from the curves – and therefore have to be provided by them. One example for this could be the introduction of a curve-dependent amount of dead shares, see issue 5.4.

Also, see issue 5.6, where we address the rounding direction of the unused preview functions.

Recommendation

Consider making `BondingCurveRegistry` and the individual curve contracts upgradeable. (Note that `BondingCurveRegistry` is `MultiVault`'s central point of communication with the curves and has to forward calls accordingly.) This will not introduce new trust assumptions, since the curve registry used can be changed in `MultiVault` anyway (and `MultiVault` itself is upgradeable too), but might make life easier. Of course, curve contract upgrades require utmost care to not break the curve's logic.

5.15 Inconsistent `msg.sender` vs `_msgSender()` Usage

Minor ✓ Fixed

Resolution

Fixed in PR 96 (reviewed commit 909cf947458e450adf642547912e11ffd08686b1) by standardizing to `msg.sender`.

Description

The codebase inconsistently uses `msg.sender` and `_msgSender()` throughout the protocol, which creates maintenance issues and style inconsistency.

Examples

src/protocol/MultiVault.sol:L362-L363

```
function approve(address sender, ApprovalTypes approvalType) external {
    address receiver = msg.sender;
```

src/protocol/MultiVault.sol:L417

```
ids[i] = _createAtom(msg.sender, _data[i], _assets[i]);
```

src/protocol/MultiVault.sol:L427

```
_addUtilization(msg.sender, int256(_payment));
```

src/protocol/MultiVault.sol:L673

```
if (!_isApprovedToDeposit(_msgSender(), receiver)) {
```

src/protocol/MultiVault.sol:L714

```
shares[i] = _processDeposit(_msgSender(), receiver, termIds[i], curveIds[i], assets[i], minShares[i]);
```

Recommendation

Consider settling on `msg.sender` or `_msgSender()` as appropriate across the codebase.

Appendix 1 - Files in Scope

This review focused on the

0xIntuition/intuition-contracts-v2 repository, revision `2bb93d259c33d6d32843655ff936efc6bec421a9`.

The following files were in scope:

File	SHA-1 hash
src/protocol/MultiVault.sol	`781ee92d8a7b4319017b92cf6132a794c0df6a90`
src/protocol/MultiVaultCore.sol	`8bce1b1dbafa4f98a62a592edea93dd92346f0ba`
src/protocol/MultiVaultMigrationMode.sol	`a1ae6a539f674b911549ceac99c579393254f7be`
src/protocol/curves/BaseCurve.sol	`c315aecce9f4b3fa999224410c5950fc80223b7c`
src/protocol/curves/BondingCurveRegistry.sol	`ec34e2ed252272c7807bc75f4eff5b4bb4326d66`
src/protocol/curves/LinearCurve.sol	`0aaa4855d57991c53d68992cac85e4b33d28590d`
src/protocol/curves/OffsetProgressiveCurve.sol	`2ad10c73e25bd4894759be96a67a0cf71d6200ec`
src/protocol/curves/ProgressiveCurve.sol	`387fefe2f47f3d94b18fb1253de09c5e62e96cba`

Appendix 2 - Disclosure

Consensys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.