Everscalend functional specification

By OCamlPro

March 19, 2022

Contents

Chapter 1

Introduction

This document contains a functional specification of the "Everscalend" smart contracts. The source code is available at
`https://github.com/SVOIcom/everscalend-contracts`, commit
`8d24e268f9c44bd3e896fb6a28bbf8a42c7027a9`. This work is provided as a submission to the Formal Methods Sub-Governance Contest #39.

Chapter 2

High-level system description

2.1 System purpose

Everscalend is a DEFI (DEcentralized FInance) lending and borrowing system implemented on the Everscale Blockchain. Its main purpose is to provide Everscale users with a realiable way to lend and borrow cryptocurrency tokens. It makes it possible for users to generate profits on tokens they supply for lending and to acquire temporarily tokens by borrowing them instead of buying them.

2.2 Terms of the system domain

Term	Definition
Interest rate	The rate of profit that is generated for the suppliers of tokens. The interest rates in Everscalend are algorithmically calculated and they increase when the borrowing demand increases and decrease when it decreases.
Market	A pool of tokens of the same kind where all the tokens supplied by the users are stored. It also holds information like the exchange rate and reserve factor etc.
vToken	A virtual token, it is a currency that only exists on Everscalend and it is used to determine the amount of tokens that a supplier owns in a market. vTokens are acquired by the users when they supply some tokens to the market. When interest rate is accumulated on the supplied tokens to the market, the exchange rate from the vTokens to the real tokens increases to account for the accumulated interest that the suppliers will get to retrieve when they withdraw their tokens.
Collateral	An amount of vTokens that a borrower has to have in order to borrow some amount of real tokens that. Everscalend like other DEFI lending systems is over-collateralized which means that the collateral has to be worth more than the amount of tokens that will be borrowed.

Term	Definition
Collateral factor	Ranging from 0 to 1, it represents the amount of tokens that can be borrowed for a collateral. e.g. a collateral factor of 0.9 allows the borrowing of a number of tokens worth 90% of the collateral.
Account health	The sum of the USD value of a user's supplied tokens divided by the sum of the USD value of all the tokens they borrowed. It is used to determine whether a user is eligible for liquidation or not. When the account health is greater than one, we say that the user's account is healthy, otherwise it is unhealthy.
Liquidation	The process in which some user's debt is liquidated by another user, by paying a portion of the owed tokens in exchange for the borrower's vTokens at a better exchange rate than the market.
Liquidation Multiplier	A value superior to 1. It is the amount by which the amount of vTokens that the liquidator should get by market price is multiplied to increase the amount they really get.
Reserve factor	Ranging from 0 to 1, it represents the portion of the interest rate that should be stored in the reserve whenever it is acquired.
Reserve	A protected portion of the accumulated interest rate on the supplied tokens. It serve as a protection of the suppliers in case they do some borrowing and become liquidable. The liquidators will only be able to take the tokens that aren't part of the reserve (Which is referred to as the cash).
Index	Refers to the interest rate index, which is a value that captures the history of interest rates of a market. It is updated after each transaction to compound the interest since the previous index.

2.3 Mathematical notations

The following mathematical notations are used in the document to simplify the equations.

$vT(T)$:	Type of vToken that have T as an underlying token.
$ER(T_1, T_2)$:	Exchange rate from the token T_1 to the token T_2, aka the x in: 1 $T_1 = x T_2$.
$TBA(T)$:	The amount of borrowed tokens in then market of the token T.
$C(T)$:	Amount of cash in the market of the token T.
$UR(T)$:	Utilization ratio of the market of the token T
$UM(T)$:	Utilization multiplier of the market of the token T.
$UbR(T)$:	Utilization base rate of the market of the token T.
$BIR(T)$:	Borrowing interest rate of the market of the token T.
$RF(T)$:	Reserve factor of the market of the token T.
$rTB(T)$:	Real token balance of the market of the token T.
$vTB(T)$:	vToken balance of the market of the token T.
$Res(T)$:	Amount of tokens stored in the reserves of the market of the token T.
$CF(T)$:	Collateral factor for a token T.
$LMul(T)$:	Liquidation multiplier of the market of the token T.
$AH(u)$:	A user u's account health.
$BC(u)$:	A user u's borrowing capacity.
$SI(u)$:	A user u's supply information.
$BI(u)$:	A user u's borrowing information.
$USA(u, T)$:	A user u's supplied amount of tokens of type T.
$USA(u, T)$:	A user u's borrowed amount of tokens of type T.
$TSAV(u)$:	Value in USD of a user u's total supplied amount.
$TBAV(u)$:	Value in USD of a user u's total borrowed amount.
$uvTA(u, T)$:	Amount of vTokens that a user u owns which have as an underlying token T.
$Ind(T, n)$:	nth interest rate index of the market of the token T.
$Indl(T)$:	Latest interest rate index of the market of the token T.
$SIF(T)$:	Simple interest factor of the market of the token T.
$AI(T)$:	Accumulated interest of the market of the token T.

2.4 System functioning

The Everscalend system uses markets, which are pools of fungible tokens with algorithmically calculated interest rates, based on the supply and borrow demand for the tokens they hold. Each market is unique to a cryptocurrency, and contains a transparent and publicly-inspectable ledger, with a record of all transactions and historical interest rates.

The lenders and the borrowers of tokens interact directly with the system, earning and paying a variable interest rate, without having to negotiate terms such as the interest rate or the value of collateral with a peer or counterparty.

In this section we describe how the system works.

2.4.1 Groups of users

A user may be part of three different groups of users with different rights:

- The admin group (or `owner` as there is only one user in the group per contract) who is the deployer of the main contracts of the system like `MarketAggregator`, `WalletController` or `Oracle`. Being part of this group gives the right to modify all the parameters of the system, and in particular to assign users to groups.
- The `Upgrader` group: being part of this group gives the right to upgrades the code of the contracts.
- The `Parameter changer` group: being part of this group gives them the to modify some of the parameters of the system.

All other users can only interact with the system to realise the market opeartions (supply, borrow ...) without being able to manually change the parameters of the system.

2.4.2 User capabilities

In this section we describe what the users can do by using the system.

2.4.2.1 Supply

A user that wishes to make a certain amount of their tokens available for borrowing has to supply them to the system. The supplied tokens are then aggregated to the tokens of the same kind that were supplied by other users.

The supplier receives vTokens for their supply that they can use as collateral to borrow other tokens. vTokens are a currency used to represent how many of the real tokens in the market a user can withdraw, they also determine how many he can borrow of other tokens.

2.4.2.2 Withdraw

A user who owns vTokens can pay with them to withdraw the tokens he supplied. The user can do it at any time provided that their account is healthy, aka $AH(u) > 1$ with $AH(u)$ being the user u account health

2.4.2.3 Borrow

To borrow a certain amount of some token:

- There have to be enough tokens in the market for the borrowing.
- The borrowed amount has to be within the user's borrowing capacity, noted $BC(u)$ with u being the user.

$BC(u)$ represents the amount of vTokens that the user can still use as collateral to borrow. Once that amount is reached, the user can no longer perform any action that requires them to have free vTokens like borrowing or withdrawing.

2.4.2.4 Repay

A borrower can repay the amount they borrowed by returning it to the market. By doing that the collateral they set for the borrowing is freed and can be used to borrow other tokens.

2.4.2.5 Liquidate

When a user's account is unhealthy, aka $(AH(u) < 1)$, the liquidation of his debt becomes possible. The liquidation process consists of selling the borrower's collateral vTokens at a discount in exchange for the repayment of the borrower's debt or a portion of it.

The purpose of this mecanism is to financially incentivize the liquidators to add liquidity to the system and pay other users' debts.

2.4.3 Key system algorithms

2.4.3.1 Interest acquisition

The values of interest rates increase when the demand is high and decrease when it is low. The calculation of the real interest requires some intermediary variables:

- The utilisation rate $UR(T)$ for the market of the token T:

$$UR(T) = \frac{TBA(T)}{rTB(T) + TBA(T)}$$

  The utilisation rate unifies supply and borrowing demand into a single variable.

- The borrowing interest rate $BIR(T)$ for the market of the token T:

$$BIR(T) = UR(T) * UM(T) + UbR(T)$$

- the simple interest factor $SIF(T)$ for the market of the token T:

$$SIF(T) = BIR(T) \times \Delta t$$

The accumulated interest for the market of the token T can then be calculated as follows:

$$AI(T) = TBA(T) \times SIF(T)$$

with Δt being the time difference in seconds between the current time and the last moment at which the interest rate was calculated.

2.4.3.2 Interest rate indexes

Each time the interest rate is calculated in a certain market, the interest rate index associated to that market is updated. New indexes are calculated as follows:

$$Ind(T, n) = (SIF(T) + 1) \times Ind(T, n - 1)$$

With: $Ind(T, 0) = 1$.

2.4.3.3 Reserves

When interest is accumulated, a portion of it, the size of which is determined by the reserve factor, is put in a reserve, the rest of it is stored as cash. After accumulating the interest, the reserve is updated by adding $(AI(T) \times RF(T))$ to it.

The purpose of the reserves is to protect a portion of the lenders interest so that they don't lose everything in case of liquidation, only their cash can be liquidated.

2.4.3.4 vToken exchange rate calculation

As stated previously the exchange rates from vTokens to real tokens depend on the supply and borrowing demand of those tokens. It is calculated as follows:

$$ER(vT(T), T) = \frac{rTB(T) + TBA(T) - Res(T)}{vTB(T)}$$

2.4.3.5 Account health and borrow capacity calculation

A user u's account health $AH(u)$ determines whether or not it is liquidable, if $AH(u) < 1$ then it is otherwise it isn't. If the user's account is liquidable then the user can't withdraw their tokens or borrow more tokens before some other users liquidiates their debt or they supply more tokens to the market to improve their account's health. The account's health is calculated whenever the user tries to perform any market operation or gets one of his loans liquidated.

Calculating $AH(u)$, first requires the calculation of The value of the user u's supplied amount $TSAV(u)$ and the value of his borrowed amount $TBAV(u)$ as follows:

$$TSAV(u) = \sum_{T} \frac{uvTA(u, T) \times ER(vT(T), T)}{ER(T, USD)} \times CF(T)$$

And

$$TBAV(u) = \sum_{(T,ba,ind) \in BI(u)} bval(T, ba, ind)$$

With:

$$bval(T, ba, ind) = \begin{cases} 0 & ba = 0 \\ \dfrac{(ba \times ind)/Indl(T)}{ER(T, USD)} & ba \neq 0 \wedge Indl(T) \neq ind \\ \dfrac{ba}{ER(T, USD)} & ba \neq 0 \wedge Indl(T) = ind \end{cases}$$

Where $BI(u)$ is the borrow information for the user u, which is in the form of a collection of triplets containing the type of the borrowed token, the amount that was borrowed and the interest rate index at the moment of the borrowing.

The user's account health is then calculated as follows:

$$AH(u) = \frac{TSAV(u)}{TBAV(u)}$$

A user's borrowing capacity is a value in USD that determines how many vTokens a user can use as collateral or redeem, the limit being that the vToken's worth has to be less that the user's borrowing capacity. The borrowing capacity is calculated as follows:

$$BC(u) = TSAV(u) - TBAV(u)$$

2.4.3.6 Liquidation

The amount of vTokens the liquidator gets is calculated thanks to the liquidation multiplier $LMul(T)$ which is the value by witch the amount of tokens they would get at market price is multiplied to increase it. If a liquidator wants to repay a portion RP of a borrower u's debt in tokens of type T, the amount of the borrower's vTokens that the liquidator will get vTA is calculated as follows:

$$vTA = min(RP \times ER(T, vT(T)) \times LMul(T), uvTA(u, T))$$

Where *min* is a function that returns the minimum of two values. The value of the liqudation multiplier is set manually by the admin.

2.4.3.7 Module locking mecanism

To avoid messing up with the pools of tokens and market parameters while a user is performing an operation, a locking mecanism is used, that prevents the modification of the data that is used during the user's operation. The locks are especially necessary when try to extract tokens from the markets by borrowing or withdarwing, or from other users through liquidation.

Such locks are used in:

- `BorrowModule`: The lock prevents other users from borrowing at the same time.
- `LiquidationModule`: In this case the lock is on the liquidated user, the purpose is to prevent more than one user liquidating the same user's debt.
- `WithdrawModule`: To prevent other users from withdrawing while one of them has started that process.
- `UserAccount`: This conract uses two locks called `borrowLock` and `liquidationLock`. As their names suggest, `borrowLock` locks the user's account during the processing of a borrow request, stopping the user from doing another one before the current one is finished. The `liquidationLock` stops the user from withdarwing or borrowing while he is being liquidated.

2.5 Architecture of the System

Figure 2.1 shows a simplistic representation of the architecture of the system. The main smart contracts are shown with their names only while the interfaces, libraries and the smart contracts that are not necessary for comprehension were omitted. The smart contracts are connected with arrows which are meant to show interactions between them, these interactions will be described below.

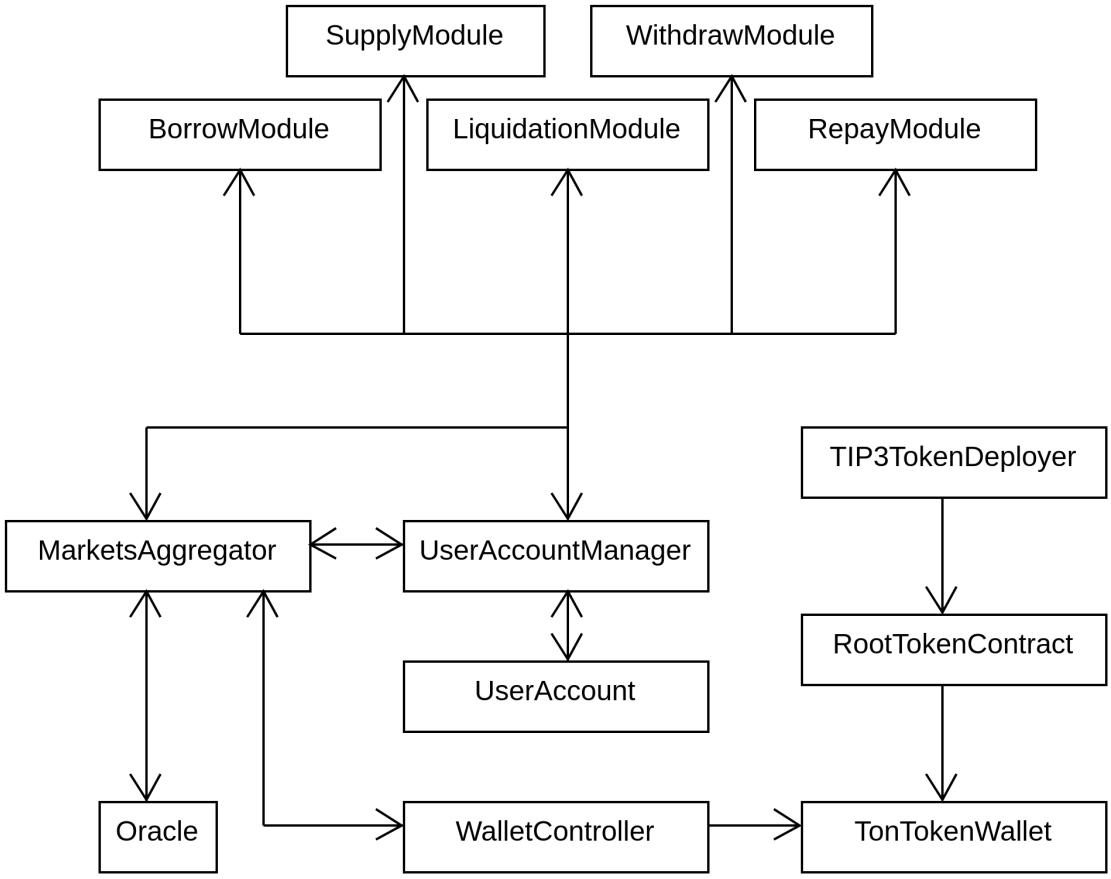

Figure 2.1: System architecture with smart contract interactions

2.5.1 Main smart contracts

2.5.1.1 MarketsAggregator

It mainly serves as a container for all the necessary information surrounding the markets, like the interest rates, the exchange rates and the borrow and supply amounts of each market.

Functionalities:

- Adding and removing markets.
- Updating the information on the markets. As a result of market operations from users (like supplying, borrowing ... etc), changes in the prices of the supported tokens or market parameter modifications by the admin or privileged users.

Interactions:

- Oracle: to get token price updates.
- Operations modules: to perform market operations.

- UserAccountManager: to update account health after performing operations.
- WalletController: to transfer tokens to(from?) when necessary.

2.5.1.2 Operations modules

These smart contracts are used to help perform the market operations by doing the necessary math for these operations.

The smart contracts:

- SupplyModule
- WithdrawModule
- BorrowModule
- RepayModule
- LiquidationModule

2.5.1.3 Oracle

It is used by the MarketsAggregator smart contract to get price updates on the supported tokens in the markets.

2.5.1.4 TonTokenWallet

It serves as a TIP-3 token wallet and it handles token transfers to and from the wallet. Everscalend follows the TIP-3 token standard and these wallets are necessary to interact with the system. They can be delpoyed either by RootTokenContract or a user who has the address of the RootTokenContract associated to the wallet's token.

Functionalities:

- Tranfer tokens to and from the wallet.
- Deploy a receiver's TonTokenWallet and transfer tokens to it.
- Burn tokens.

2.5.1.5 RootTokenContract

A contract that is deployed by a token owner and stores information about the root token (like its name, symbol and total supply).

Functionalities:

- Minting tokens.
- Deploying instances of TonTokenWallet.
- Updating the information on the root token.
- Change root token owner.

2.5.1.6 TIP3TokenDeployer

it deploys RootTokenContract contracts.

2.5.1.7 WalletController

It's the smart contract that controls all the TIP-3 wallets of the markets and managers the operations that require token transfers to and from those wallets.

Functionalities:

- Adding and removing market wallets.
- Getting information about market wallets.
- Decoding the payloads of messages that require TIP-3 token transfers and adds necessary information like message origin and token amount before passing them to MarketsAggregator.
- Allows the users to perform the supply, repay and liquidate operations.

Interactions:

- MarketsAggregator: to communicate to it the details of the market operations that require TIP-3 token transfer into the wallets of the markets.
- TonTokenWallet: to transfer tokens a user's wallet.

2.5.1.8 UserAccount

It's used to store information about how a user interacts with the markets, like how much they borrowed from which market and how much they supplied to which market. It also allows the user to perform the borrow and withdraw operations.

It interacts only with UserAccountManager, which can request the information and to which it returns it.

2.5.1.9 UserAccountManager

UserAccountManager serves as an intermediary between the user, the UserAccount contract, the MarketsAggregator contract and the market operations modules.

Functionalities:

- Deploys UserAccount contract.
- Handles requests and data transfers from MarketsAggregator and market modules to the UserAccount contract.

Interactions:

- MarketsAggregator: to calculate user account health and perform market operations that were requested by the user.
- market modules: to perform market operations.
- UserAccount: to get user information and to update it.

2.5.2 User interactions

The users mainly interact with TonTokenWallet and UserAccount to perform the market operations mentioned previously. The operations in which the users have to request tokens are done through the UserAccount (withdraw, borrow) smart contract, while the operations in which the users have to send tokens are done through TonTokenWallet (supply, repay, liquidate).

2.6 Usage scenarios

This section contains low level descriptions of usage scenarios which describe how the users interatct with the smart contracts and how the smart contracts interact between them.

Used notation:

UA:	UserAccount	UAM:	UserAccountManager
MA:	MarketsAggregator	WC:	WalletController
TTK:	TonTokenWallet	SM:	SupplyModule
WM:	WithdrawModule	BM:	BorrowModule
RM:	RepayModule	LM:	LiquidationModule

In the source code a lot of the interactions between the smart contracts are done with the intermediary of interfaces. In these usage scenarios we ignore the interfaces and present only the interactions between the contracts which the interfaces refer to.

2.6.1 Updating a user's account health

We start with a "sub-scenario" which is the one of updating the user's account health. This scenario was added because it's used by the other ones and to avoid repeating it everytime, we describe here what happens when we that the user's account health is updated.

1. `UAM.calculateUserAccountHealth` is called from |UA| with a payload.
2. `UAM.calculateUserAccountHealth` calls `MA.calculateUserAccountHealth` which:
   2.1. Updates the markets' information and calculates the user's account health.
   2.2. If the user's account is unhealthy a `LiquidationPossible` event is emitted.
   2.3. A call is made to `UAM.updateUserAccountHealth` with the new account health.
3. `UAM.updateUserAccountHealth` calls `UA.updateUserAccountHealth` which calls, depending on the operation code provided in the payload, either `UAM.requestTokenPayout`, `UAM.returnAndSupply` or transfers the remaining gas to a povided address.

2.6.2 Supply

1. The user makes an internal transfer through `TTK.internalTransfer` with a payload containing the supply operation code.
2. The payload with information about the token wallet is passed to `WC.tokensReceivedCallback`.
3. The payload is decoded and after checking the operation code a call is made to `MA.performOperationWalletController` with the necessary information about the supply operation.
4. `MA.performOperationWalletController` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.
5. `MA.performOperation` calls `SM.performAction` which:
   5.1. Calculates the amount of vTokens to provide the user.
   5.2. Builds a data structure with the changes to the market (market delta) to which the supply was made.

6. The market delta is sent to `MA.receiveCacheDelta` which updates market informations and calls `SM.resumeOperation` with the new market information.
7. The information about the supply operation is then sent to `UAM.writeSupplyInfo` which transfers it to `UA.writeSupplyInfo`.
8. `UA.writeSupplyInfo` calls `UAM.calculateUserAccountHealth` with a payload having the `NO_OP` operation code.
9. After updating account health, the remaining gas is transferred back to the supplier.

2.6.3 Withdraw

1. The user calls `UA.withdraw` with the address of their TIP-3 wallet, the ID of the market to which their supplied their tokens and the amount their wish to withdraw.
2. `UA.withdraw` calls `UAM.requestWithdraw` with the withdrawal information.
3. `UAM.requestWithdraw` calls `MA.performOperationUserAccountManager` with the withdraw operation code.
4. `MA.performOperationUserAccountManager` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.
5. `MA.performOperation` calls `WM.performAction`.
6. `WM.performAction` locks the module `WM` and requests from `UAM` the user's borrow and supply information which is provided by `UA` and goes through `UAM.receiveWithdrawInfo` which sends it to `WM.withdrawTokensFromMarket`.
7. `WM.withdrawTokensFromMarket` checks that:
   - The user's account is healthy.
   - The user supplied at least as many tokens as the amount that they wish to withdraw.
   - The worth in USD of the user's free collateral is equal or greater than the worth in USD of the amount of tokens they wish to withdraw.

   Then calls `MA.receiveCacheDelta` with the withdrawal information.
8. `MA.receiveCacheDelta` updates market information and calls `WM.resumeOperation` with the new market information.
9. `WM.resumeOperation` unlocks the module `WM` and sends the withdrawing information to `UAM.writeWithdrawInfo` which calls `UA.writeWithdrawInfo` which:
   9.1. updates the user's supply information by decreasing it with the withdrawn amount.
   9.2. builds a payload with the amount of tokens that need to be sent, the user's TIP-3 wallet address and the opeartion code `REQUEST_TOKEN_PAYOUT`.
10. A call is made to `UAM.calculateUserAccountHealth` with the user's new supply and borrow information.
11. After the user's account health is updated, `UAM.returnAndSupply` is called.

12. `UAM.returnAndSupply` calls `MA.requestTokenPayout` and `WM.unlock` (which does nothing because the module was unlocked during the call to `WM.resumeOperation`).
13. `MA.requestTokenPayout` calls `WC.transferTokensToWallet` which calls `WC.transfer`.
14. `WC.transfer` calls `TTK.internalTransfer` which updates the user's balance with the amount of tokens that he requested to withdraw.

2.6.4 Borrow

1. The user calls `UA.borrow` with the address of their TIP-3 wallet, the ID of the market from which they wish to borrow and the amount of tokens they wish to borrow.
2. `UA.borrow` locks the user account and calls `UAM.requestIndexUpdate` with the borrowing information.
3. `UAM.requestIndexUpdate` calls `MA.performOperationUserAccountManager` with the `BORROW_TOKENS` operation code.
4. `MA.performOperationUserAccountManager` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.
5. `MA.performOperation` calls `BM.performAction`.
6. `BM.performAction` locks the module `BM` and requests from `UAM` to update the market's indexes which is done by calling `UA.borrowUpdateIndexes`.
7. `UA.borrowUpdateIndexes` gets from the markets the updates indexes and passes them along with the user's borrow and supply into to `UAM.passBorrowInformation` which calls `BM.borrowTokensFromMarket`.
8. `BM.borrowTokensFromMarket` checks that:
   - That there are enough tokens in the market for the borrowing.
   - That the user's account is healthy.
   - That the user has enough collateral to make the borrowing.
9. A `TokenBorrow` event is emitted with the borrowing information.
10. `BM.borrowTokensFromMarket` calls `MA.receiveCacheDelta` with the borrowing information and the information about the changes to the market after the borrowing.
11. `MA.receiveCacheDelta` updates the market information and calls `BM.resumeOperation` with the new market information.
12. `BM.resumeOperation` unlocks the module `BM` and sends the borrowing information to `UAM.writeBorrowInformation` which calls `UA.writeBorrowInformation` which:
    12.1. Updates market information and the user's borrowing information.
    12.2. Unlocks `UA`.
    12.3. Builds a payload with the `REQUEST_TOKEN_PAYOUT` operation code.

13. `UAM.calculateUserAccountHealth` is called with the payload as well as the user's supply and borrow information.
14. After the user's account health is updated, `MA.requestTokenPayout` is called.
15. `MA.requestTokenPayout` calls `WC.transferTokensToWallet` which calls `WC.transfer`.
16. `WC.transfer` calls `TTK.internalTransfer` which updates the user's balance with the amount of tokens that they borrowed.

2.6.5 Repay

1. The user makes an internal transfer through `TTK.internalTransfer` with a payload containing the repay operation code.
2. The payload with information about the token wallet is passed to `WC.tokensReceivedCallback`.
3. The payload is decoded and after checking the operation code a call is made to `MA.performOperationWalletController` with the necessary information about the supply operation.
4. `MA.performOperationWalletController` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.
5. `MA.performOperation` calls `RM.performAction` which calls `UAM.requestRepayInfo`.
6. `UAM.requestRepayInfo` calls `UA.sendRepayInfo` which updates market information and transfers it with information about the user's TIP-3 wallet to `UAM.receiveRepayInfo` which calls `RM.repayLoan`.
7. `RM.repayLoan` calculates how much of the loan will be repayed and the changes to the markets after the repayment. A `RepayBorrow` event is emitted with the repayment information. That information is then sent to `MA.receiveCacheDelta` which updates market informations and calls `RM.resumeOperation` with the new market information.
8. `RM.resumeOperation` calls `UAM.writeRepayInformation` which transfers it to `UA.writeRepayInformation`.
9. `UA.writeRepayInformation` calls `UAM.calculateUserAccountHealth` with one of the two operation codes:
   - `REQUEST_TOKEN_PAYOUT` if there are leftover tokens after the repayment.
   - `NO_OP` otherwise.
10. After updating account health, if there are leftover tokens after the repayment, they are transferred to the user's TIP-3 wallet.

2.6.6 Liquidate

1. The user makes an internal transfer through `TTK.internalTransfer` with a payload containing the liquidation operation code.
2. The payload with information about the token wallet is passed to `WC.tokensReceivedCallback`.
3. The payload is decoded and after checking the operation code a call is made to `MA.performOperationWalletController` with the necessary information about the liquidation.
4. `MA.performOperationWalletController` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.
5. `MA.performOperation` calls `LM.performAction` which calls `UAM.requestLiquidationInformation`.
6. `UAM.requestLiquidationInformation` calls `UA.requestLiquidationInformation` which updates market's indexes and calls `UAM.receiveLiquidationInformation` with the new indexes and the user's supply and borrow information.
7. `UAM.receiveLiquidationInformation` calls `LM.liquidate` which:
   (a) Checks the account health of the user that is targeted for liquidation to check that it is still required.
   (b) Selects the minimum between the provided amount of tokens for the liquidation and the borrowed amount by the targeted user as the liquidation amount.
   (c) Calculates the USD value of the liquidation amount.
   (d) Calculates the how many of the targeted user's collateral to seize.
   (e) Emits a `TokensLiquidated` event with the liquidation information, update the markets and the liquidated user's borrow information.
   (f) Calls `MA.receiveCacheDelta` which updates market informations and calls `RM.resumeOperation` with the new market information.
8. `LM.resumeOperation` recovers the sent information and passes it to `UAM.seizeTokens` which calls `UA.liquidateVTokens`.
9. `UA.liquidateVTokens` updates the user's borrow and supply information and calls `UAM.grantVTokens`.
10. `UAM.grantVTokens` calls `UA.checkUserAccountHealth` on the target user's account and wallet addresses to check their account health. Also calls `UA.grantVTokens`.
11. `UA.grantVTokens` checks the liquidator's account health and builds a payload with the operation code `RETURN_AND_UNLOCK` to pass to the function that checks the user's account.
12. Once the new user's account health is recovered `UAM.returnAndSupply` is called.
    - if there are leftover tokens after the liquidation then a call is made to `MA.requestTokenPayout` with the tokens to return and the liquidator's TIP-3 wallet.
    - A call is made to `LM.unlock` to unlock it and return the remaining gas to the liquidator's wallet.

13. `MA.requestTokenPayout` calls `WC.transferTokensToWallet`.

14. A call is then made to `TTK.transfer` which calls `TTK.internalTransfer` to update the user's balance by increasing it with the returned amount of tokens.

Chapter 3

Risks

In this chapter we present the potential risks that can threaten the Everscalend System. Some of these risks are more general to DEFI systems, some others are specific to the Everscalend system. We separate the risks by type into two categories, financial risks, which originate from the market mechanics of the system and smart contract risks, which are the risks that are usually present in smart contract source code.

3.1 Financial risks

3.1.1 Insolvency

Insolvency is when a borrower's loans become worth more than their collateral. In this case neither they nor the liquidators are incentivized to repay the loan, which removes liquidity from the market since these users will hold on to loans that will not be repaid. Insolvency can happen if the underlying tokens of the collateral vTokens lose their value quickly or the borrowed tokens' price increases rapidly.

3.1.2 Illiquidity

Illiquidity is when there aren't enough tokens in the market for a supplier to do a withdrawal or for a borrower take out a loan. It is problematic because users are supposed to have control over their tokens and be able to withdraw them whenever they want to, given that their account's health allows it. Illiquidity can happen if the price of the borrowed token increases rapidly also, that will disincentivise the borrowers and the liquidators from repaying the loan as they would rather keep holding their tokens or selling them. Same as the suppliers which will lead to a bank run (an event in which suppliers will try to withdraw their supplies as quickly as possible to avoid losing tokens) and that will lead to the slowest suppliers, especially if they want to borrow big amounts, to lose some or all of their tokens since the loans aren't getting repaid.

3.1.3 Unfair liquidation

To determine whether or not a user's borrowings should be liquidated. The Everscalend system checks if their account is healthy. If it isn't then they can be liquidated. The issue is that if a user has only one borrowing in which the borrowed tokens price jumps quickly and decreases their borrowing capacity to zero. All their borrowings become liquidable. Which makes it possible

for liquidators to target only the cheapest ones, to make sure that the borrower's account stays unhealthy for as long as possible, so they can profit off of it, which could be considered unjust for the borrower especially if they have many borrows and only one that causes their account to be unhealthy. This will make the users weary of it happening to them and less likely to make many borrowings, especially big ones.

3.1.4 Centralization

The risk that comes with having an administrator or super user role which gives the detainer of that role the capability to unilaterally and arbitrarily modify the functioning of the system is in itselft risky. Especially since many values like the collateral factor and liquidation multiplier are editable with an admin role, he can also decide who can and who can't change market parameters. Therefore control over those parameters and who and how they can be changed need to be clarified. There needs to be a guarantee that one super user can't manually and unilaterally modify the entire functioning of the system.

3.2 Smart contract risks

3.2.1 Unsound math

Math operations use approximations and rounding. It could lead in some particular cases to errors that could affect the functioning of the system or introduce vulnerabilities that can be taken advantage of.

3.2.2 Non liquidation

To determine whether a user's loans can be liquidated or not, his account health has to be calculated, it if his account is unhealthy a notification is send to the system informing the other users of that. In Everscalend the user 's account health is calculated whenever they try to perform some operation. If the checks are not regularly and externally done, a user who does not perform any operation for a while, can end up having an unhealthy account without it being notified to the other users of the system, which could lead to the liquidation not happening. A user can also wait without doing any operation, until their account health raises back again or they get enough tokens to suplly to the market to raise it by themselves.

3.2.3 Locking

Using locks in programs is sometimes necessary but it is always tricky. The developers have the make sure that locks are locked and unlocked at the right times. Otherwise there are various risks like data corruption or permanently locking some code and making it unusable. There is also the risk of the locks taking too long to be unlocked, making the system less performant.

3.2.4 Visibility

It involves all the risks of having functions in contracts that are accessible to users which are not intended to use them. Especially if they are functions that write information on the system and significantly affect it's functioning.

Chapter 4

System Properties

This chapter lists the system properties that mend for some of the risks presented in the previous section. The mathematical notations defined in 2.3 are used.

Chapter 5

Code audit

This chapter presents an audit of Everscalend's smart contracts and lists the issues that were encoured in the source code.

Table of all Found Issues

5.1 General remarks

In this section we present some recurrent issues that were encountered in the source code and some general good practices that should be respected.

5.1.1 Typography of Internal Functions

A good coding convention is to use typography to visually discriminate public functions and internal functions, for example using a prefix such as `_`.

5.1.2 Constructors without checks

Contract constructors should always at the very least verify that the contract's public key is set and that the deployer is the owner of the contract. This is important especially in the case in which the contract has arguments that set the state variables. If it is not done, it opens the gate to various kinds of attacks.

5.2 Contract deployment from Platform

Critical issue: Unprotected constructors in many contracts
See 5.1.2. Other than the 'RootTokenContract' and 'TONTokenWallet' contracts, all the other contracts have unprotected constructors and a comment that says that the contract will be deployed from the Platform. That does mean that is no longer necessary to check that the deployer of the contract is the owner of the contract. It is especially dangerous in the contracts which set the owner through the constructor like: 'MarketAggregator', 'BorrowModule', 'LiquidationModule', 'RepayModule', 'SupplyModule', 'WithdrawModule', 'Oracle', 'TIP3TokenDeployer', 'UserAccount', 'UserAccountManager', 'Platform' and 'WalletController'.

5.2.1 Possible attack

It makes it possible to perform phishing attacks by deploying fake contracts with which the users can interact with. So instead of interacting with the real contract they interact with the fake.

If a malicious user deploys a fake 'UserAccountManager' which will deploy user's accounts. And one of the users requests a withdraw of their tokens. The owner of the fake 'UserAccountManager' can either block the transaction stopping the user's from withdrawing their tokens, or ask them for a fee before processing their request.

5.3 Unsafe role assignement

Critical issue: Unsafe role assignement in `IRoles.sol`
The role setter functions `setUpgrader` and `setParamChanger` in the `IRoles` abstract contract (file: "IRoles.sol") allow the contract owner to assign the upgrader or paramChanger role to the provided address or unassign it. The function should test that the provided address's value is not zero. Otherwise the owner can mistakingly give that right to all external users, which would make it possible for anyone to modify the functioning of the system.

5.4 Internal function names

Minor issue: Internal function names

See 5.1.1.

The functions `performOperation` and `updatePrice` in the `MarketAggregator` contract (file: "MarketsAggregator.sol"). are internal so their names should start with '_'.

5.5 Undefined functions

Minor issue: Undefined functions

The functions `calculateUtilizationRate`, `calculateBorrowingRate` and `calculateExchangeRate` in the 'MarketMath' library (file: "MarketMath.sol") are undefined and unused. Defining and using them appropriately would significantly improve the readability of the source code.

5.6 Unused functions

Minor issue: Unused functions

The functions:

- `calculateExchangeRate` in the `MarketMath` library (file: "MarketMath.sol")
- `calculateU`, `calculateTotalReserves`, `calculateNewIndex`, `calculateTotalBorrowed` and `calculateReserves` in the `MarketOperations` library (file: "MarketOperations.sol")
- `_calculateBorrowInfo` in the `MarketAggregator` conract (file: "MarketsAggregator.sol")

And all the functions from the `MarketToUserPayloads` library (file: "MarketPayloads.sol") and `TvmCellOperations` library (file: "TvmCellOperations.sol") are unused. Unused functions should be removed from the code as they are useless and they clutter the code.

5.7 Unused modifiers

Minor issue: Unused modifiers

The modifiers:

- `onlySelf`, `onlyRealTokenRoot` and `onlyExecutor` in the `MarketAggregator` contract (file: "MarketsAggregator.sol")
- `onlyMarket` in the `WalletController` conract (file: "WalletController.sol")

Are unused. Unused modifiers should be removed from the code as they are useless and they clutter the code.

5.8 Library MarketMath

In file `MarketMath.sol`

5.8.1 Function calculateExchangeRate

```
14      function calculateExchangeRate(uint256 currentPool, uint256 totalBorrowed,
            uint256 totalReserves, uint256 totalSupply)
15          internal pure returns (uint256)
16      {
17          return math.div(currentPool - totalReserves + totalBorrowed, totalSupply);
18      }
```

Minor issue: Syntax Error in `MarketMath.calculateExchangeRate`
`math.div` does not exist. It should use the infix division operator `/`.

5.9 Library Utilities

In file `IModule.sol`

5.9.1 Function calculateSupplyBorrow

```
90      function calculateSupplyBorrow(
91          mapping(uint32 => uint256) supplyInfo,
92          mapping(uint32 => BorrowInfo) borrowInfo,
93          mapping(uint32 => MarketInfo) marketInfo,
94          mapping(address => fraction) tokenPrices
95      ) internal returns (fraction) {
96          fraction accountHealth = fraction(0, 0);
97          fraction tmp;
98          fraction nom = fraction(0, 1);
99          fraction denom = fraction(0, 1);
100
101         // Supply:
102         // 1. Calculate real token amount: vToken*exchangeRate
103         // 2. Calculate real token amount in USD: realTokens/tokenPrice
104         // 3. Multiply by collateral factor: usdValue*collateralFactor
105         for ((uint32 marketId, uint256 supplied): supplyInfo) {
106             tmp = supplied.numFMul(marketInfo[marketId].exchangeRate);
107             tmp = tmp.fDiv(tokenPrices[marketInfo[marketId].token]);
108             tmp = tmp.fMul(marketInfo[marketId].collateralFactor);
109             nom = nom.fAdd(tmp);
110             nom = nom.simplify();
111         }
112
113         // Borrow:
114         // 1. Recalculate borrow amount according to new index
115         // 2. Calculate borrow value in USD
116         // NOTE: no conversion from vToken to real tokens required, as value is
                stored in real tokens
117         for ((uint32 marketId, BorrowInfo _bi): borrowInfo) {
118             if (_bi.tokensBorrowed != 0) {
119                 if (!_bi.index.eq(marketInfo[marketId].index)) {
120                     tmp = borrowInfo[marketId].tokensBorrowed.numFMul(marketInfo[
                            marketId].index);
121                     tmp = tmp.fDiv(borrowInfo[marketId].index);
122                 } else {
```

```
123                     tmp = borrowInfo[marketId].tokensBorrowed.toF();
124                 }
125                 tmp = tmp.fDiv(tokenPrices[marketInfo[marketId].token]);
126                 tmp = tmp.simplify();
127                 denom = denom.fAdd(tmp);
128                 denom = denom.simplify();
129             }
130         }
131
132         accountHealth = nom.fDiv(denom);
133
134         return accountHealth;
135     }
```

Minor issue: Unintuitive function name
The function is called “calculateSupplyBorrow” but it calculates a user’s account health. It should be named accordingly, e.g. “calculateAccountHealth”.

5.10 Contract BorrowModule

In file `BorrowModule.sol`

5.10.1 Function borrowTokensFromMarket

```
74      function borrowTokensFromMarket(
75          address tonWallet,
76          address userTip3Wallet,
77          uint256 tokensToBorrow,
78          uint32 marketId,
79          mapping (uint32 => uint256) supplyInfo,
80          mapping (uint32 => BorrowInfo) borrowInfo
81      ) external override onlyUserAccountManager {
82          tvm.rawReserve(msg.value, 2);
83          mapping(uint32 => MarketDelta) marketsDelta;
84          MarketDelta marketDelta;
85
86          // Borrow:
87          // 1. Check that market has enough tokens for lending
88          // 2. Calculate user account health
89          // 3. Calculate USD value of tokens to borrow
90          // 4. Check if there is enough (collateral - borrowed) for new token
                borrow
91          // 5. Increase user's borrowed amount
92
93          MarketInfo mi = marketInfo[marketId];
94
95          if (tokensToBorrow < mi.realTokenBalance - mi.totalReserve) {
96              fraction accountHealth = Utilities.calculateSupplyBorrow(supplyInfo,
                    borrowInfo, marketInfo, tokenPrices);
97              if (accountHealth.nom > accountHealth.denom) {
98                  uint256 healthDelta = accountHealth.nom - accountHealth.denom;
99                  fraction tmp = healthDelta.numFMul(tokenPrices[marketInfo[marketId
                        ].token]);
100                 uint256 possibleTokenWithdraw = tmp.toNum();
101                 if (possibleTokenWithdraw >= tokensToBorrow) {
102                     marketDelta.totalBorrowed.delta = tokensToBorrow;
103                     marketDelta.totalBorrowed.positive = true;
104                     marketDelta.realTokenBalance.delta = tokensToBorrow;
```

```
105                     marketDelta.realTokenBalance.positive = false;
106
107                     marketsDelta[marketId] = marketDelta;
108
109                     TvmBuilder tb;
110                     tb.store(marketId);
111                     tb.store(tonWallet);
112                     tb.store(userTip3Wallet);
113                     tb.store(tokensToBorrow);
114
115                     emit TokenBorrow(marketId, marketDelta, tonWallet,
                            tokensToBorrow);
116
117                     IContractStateCacheRoot(marketAddress).receiveCacheDelta{
118                         flag: MsgFlag.REMAINING_GAS
119                     }(marketsDelta, tb.toCell());
120                 } else {
121                     IUAMUserAccount(userAccountManager).writeBorrowInformation{
122                         flag: MsgFlag.REMAINING_GAS
123                     }(tonWallet, userTip3Wallet, 0, marketId, marketInfo[marketId
                            ].index);
124                 }
125             } else {
126                 IUAMUserAccount(userAccountManager).writeBorrowInformation{
127                     flag: MsgFlag.REMAINING_GAS
128                 }(tonWallet, userTip3Wallet, 0, marketId, marketInfo[marketId].
                        index);
129             }
130         } else {
131             address(tonWallet).transfer({value: 0, flag: MsgFlag.REMAINING_GAS});
132         }
133     }
```

Critical issue: Math error in `BorrowModule.borrowTokensFromMarket`

Line 99. To caculate the amount of tokens that it is possible to withdraw, the health delta needs to be divided by the price of the token not multiplied by it.

5.11 Module "IUserAccount.sol"

5.11.1 Struct UserMarketInfo

```
10  struct UserMarketInfo {
11      bool exists;
12      uint32 _marketId;
13      uint256 suppliedTokens;
14      fraction accountHealth;
15      BorrowInfo borrowInfo;
16  }
```

Minor issue: Unused struct field in `UserMarketInfo`

The field `accountHealth` is unused.

5.12 Contract Platform

In file `Platform.sol`.

5.12.1 Function initializeContract

```
18  function initializeContract(TvmCell contractCode, TvmCell params) private {
19      tvm.accept();
20      TvmBuilder builder;
21
22      builder.store(root);
23      builder.store(platformType);
24
25      builder.store(platformCode); // ref 1
26      builder.store(initialData);  // ref 2
27      builder.store(params);       // ref 3
28
29      tvm.setcode(contractCode);
30      tvm.setCurrentCode(contractCode);
31
32      onCodeUpgrade(builder.toCell());
33  }
```

Major issue: `tvm.accept` in a private function
Private and internal functions should not have a `tvm.accept`, especially without checks.

5.13 Module "FloatingPointOperations.sol"

5.13.1 Struct fraction

```
3   struct fraction {
4       uint256 nom;
5       uint256 denom;
6   }
```

Minor issue: Unintuitive struct field name
The name of the field `nom` should be `num` for "numerator".

5.14 Library FPO

In file `FloatingPointOperations.sol`

5.14.1 Function eq

```
53      function eq(fraction a, fraction b) internal pure returns(bool) {
54          return ((a.nom == b.nom) && (a.denom == b.denom));
55      }
```

Major issue: Math error in `FPO.eq`
Comparing numerators and denominators when testing if fractions are equal is incorrect. $eq(\frac{a}{b}, \frac{a\times 2}{b\times 2})$ will return `false` while it should return `true`. The fractions need to be normalized before testing the equality.

5.14.2 Function simplify

```
69  function simplify(fraction a) internal pure returns(fraction) {
70      // loosing ??? of presicion at most
71      if (a.nom / a.denom > 100e9) {
72          return fraction(a.nom / a.denom, 1);
73      } else {
74          // using bitshift for simultaneos division
75          // leaving up to 64 bits of information if nom & denom > 2^64
76          if ( (a.nom >= bits224) && (a.denom >= bits224) ) {
77              return fraction(a.nom / bits160, a.denom / bits160);
78          }
79
80          if ( (a.nom >= bits192) && (a.denom >= bits192) ) {
81              return fraction(a.nom / bits128, a.denom / bits128);
82          }
83
84          if ( (a.nom >= bits160) && (a.denom >= bits160) ) {
85              return fraction(a.nom / bits96, a.denom / bits96);
86          }
87
88          if ( (a.nom >= bits128) && (a.denom >= bits128) ) {
89              return fraction(a.nom / bits64, a.denom / bits64);
90          }
91
92          if ( (a.nom >= bits96) && (a.denom >= bits96) ) {
93              return fraction(a.nom / bits32, a.denom / bits32);
94          }
95
96          return a;
97      }
98  }
```

Minor issue: Math issue in `FPO.simplify`

Dividing the numerator and denominator by their greatest common divisor might make it unnecessary to do the bitshift and avoid losing precision.