Security Review & Formal Specification Report
NM-0058: ZkLend (Layer 2)

NETHERMIND

(May 23, 2022)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind on the Layer 2 core contracts of ZkLend written in Cairo language. ZkLend is a StarkNet native lending protocol where users can deposit for `ZTokens` which represents a user's portion of liquidity owned in a pool. `ZTokens` can be used as collateral to borrow from pools in the protocol, and as loans are paid back with interest the value of the pool grows alongside the value of the `ZTokens`. The liquidity in pools can also be used for flash loans, bringing in extra fees to the protocol. A liquidate feature also exists to prevent account under-collateralization. The contracts have a high degree of decentralization and pause functionalities have not been implemented. The codebase is composed of 2,953 lines of Cairo code. The application has high quality and the code follows the best smart contract programming practices. The application depends on the external oracle Empiric Network, which has not been audited up to this moment. The external oracle provides the price of assets for `ZkLend`. Having a robust application for dealing with external entities malfunctioning is very important to protect the users' assets. In this sense, `ZkLend` must be able to identify and promptly respond to unexpected events originating from external applications.

During the initial audit, we point out 12 potential issues. During the reaudit, the ZkLend team has fixed 4 issues, while 8 issues have been `acknowledged` since those issues will be addressed in the Cairo 1 version. The distribution of issues is summarized in the figure below. This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the findings in a table. Section 4 presents the formal specification developed by the Nethermind Formal Verification team. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details each finding. Section 7 discusses the documentation provided for this audit. Section 8 presents the output of the automated test suite. Section 9 concludes the audit report.

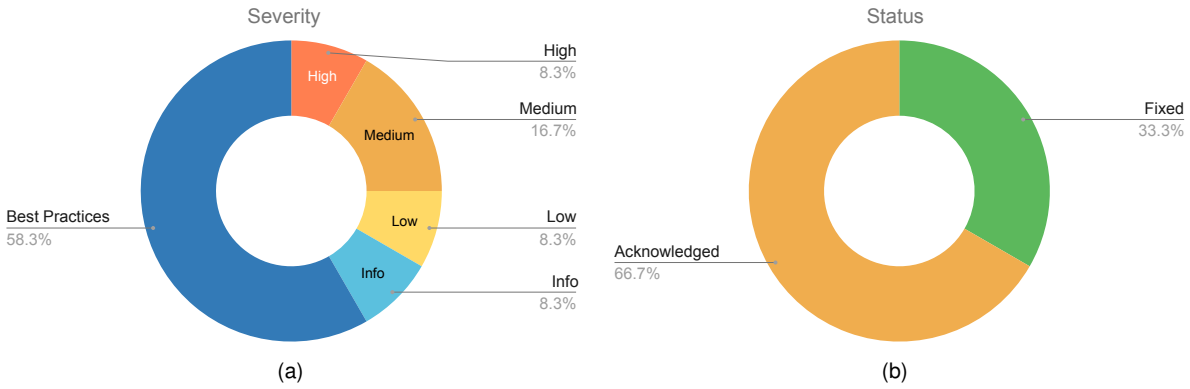

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (2), **Low** (1), **Undetermined** (0), **Informational** (1), **Best Practices** (7). **Distribution of status: Fixed** (4), **Acknowledged** (8), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Audit Report	Sep. 30, 2022
Response from Client	Apr. 10, 2023
Final Audit Report	Apr. 24, 2023
Final Formal Specification Report	Apr. 28, 2023
Methods	Manual Review, Automated Analysis
Repository	ZkLend-v1-core
Commit Hash (Initial Audit)	c7517fea032e75004f1328e7ab65ccc6f4b393cf
Documentation	Available here
Documentation Assessment	Medium
Test Suite Assessment	High

2 Contracts

	Contract	Lines of Code	Lines of Comments	Comments Ratio	Blank Lines	Total Lines
1	src/zklend/Proxy.cairo	50	10	20.0%	11	71
2	src/zklend/ZToken.cairo	155	15	9.7%	32	202
3	src/zklend/PriceOracle.cairo	36	19	52.8%	15	70
4	src/zklend/Market.cairo	174	17	9.8%	33	224
5	src/zklend/libraries/SafeMath.cairo	40	1	2.5%	10	51
6	src/zklend/libraries/SafeDecimalMath.cairo	36	6	16.7%	11	53
7	src/zklend/libraries/SafeCast.cairo	29	7	24.1%	6	42
8	src/zklend/libraries/Math.cairo	35	8	22.9%	9	52
9	src/zklend/internals/Market/structs.cairo	19	1	5.3%	1	21
10	src/zklend/internals/Market/storage.cairo	287	98	34.1%	78	463
11	src/zklend/internals/Market/events.cairo	54	2	3.7%	13	69
12	src/zklend/internals/Market/functions.cairo	1115	110	9.9%	247	1472
13	src/zklend/internals/ZToken/storage.cairo	25	1	4.0%	9	35
14	src/zklend/internals/ZToken/events.cairo	4	1	25.0%	2	7
15	src/zklend/internals/ZToken/functions.cairo	365	29	7.9%	99	493
16	src/zklend/oracles/EmpiricOracleAdapter.cairo	64	16	25.0%	18	98
17	src/zklend/irms/DefaultInterestRateModel.cairo	74	17	23.0%	23	114
18	src/zklend/interfaces/IPriceOracleSource.cairo	8	3	37.5%	3	14
19	src/zklend/interfaces/IPriceOracle.cairo	8	3	37.5%	3	14
20	src/zklend/interfaces/IZToken.cairo	49	7	14.3%	26	82
21	src/zklend/interfaces/IInterestRateModel.cairo	8	1	12.5%	2	11
22	src/zklend/interfaces/IMarket.cairo	66	10	15.2%	29	105
23	src/zklend/interfaces/third_parties/IEmpiricOracle.cairo	8	1	12.5%	2	11
24	src/zklend/interfaces/callback/IZklendFlashCallback.cairo	6	1	16.7%	2	9
	Total	**2715**	**384**	**14.1%**	**684**	**3783**

3 Summary of Issues

	Finding	Severity	Update
1	Flash Loan fees do not increase the `ZToken` accumulator value	High	Fixed
2	Application must be robust and protect itself against oracle malfunction	Medium	Acknowledged
3	No action is taken when oracle prices are outdated	Medium	Acknowledged
4	Application ignores the number of sources used for composing the exchange price	Low	Acknowledged
5	Missing check for `ZToken` market address in `add_reserve(...)`	Info	Acknowledged
6	Comment does not match function logic	Best Practices	Fixed
7	Emit events on contract deployment and initialization	Best Practices	Acknowledged
8	Missing argument validations in transfer functions	Best Practices	Acknowledged
9	Missing events emission in `set_token_source(...)`	Best Practices	Fixed
10	Missing safety features in privileged address and implementation changes	Best Practices	Acknowledged
11	Unused functions	Best Practices	Fixed
12	Variable naming for ZToken amounts can be unclear	Best Practices	Acknowledged

4 ZkLend Formal Specification Report

This report presents the formalization in the Coq proof assistant[2] of the ZkLend money-market protocol, as described in the whitepaper [3], informed by the implementation in the commit hash fe7f522966a1df3ca0c59190a1a74b2d1c8b12af.

4.1 Model

In order to specify the protocol, we must first define an abstraction to represent its state. In what follows, we describe the abstract state components and other useful notations used in our formalization.

4.1.1 State representation

The abstract state representation makes use of the following basic types:

- `Token`: The set of tokens the protocol accepts.
- `Time`: The set of time stamps.
- `User`: The set of users of the protocol. The market is represented as a special value of the `User` type.
- $\mathbb{Q}$: The set of rational numbers.
- $\mathbb{Q}^{\geq 0}$: The set of rational numbers bigger or equal to zero, i.e., $\{x \in \mathbb{Q} \mid x \geq 0\}$.
- $[0, 1] \subseteq \mathbb{Q}$: closed interval of rational numbers between 0 and 1, i.e., $\{x \in \mathbb{Q} \mid 0 \leq x \leq 1\}$.

The corresponding Coq code for basic types and operations over them are defined here.

The protocol's state, , consists of a record with members:

- $erc20\_balances$: `Token` → `User` → $\mathbb{Q}^{\geq 0}$: defines a mapping to store the token balance for a given user or the market itself.
- $erc20\_allowances$: `Token` → `User` → `User` → $\mathbb{Q}^{\geq 0}$: defines a mapping that stores the allowances from a token from a user or market to another user or the market itself.
- $z\_balances$: `Token` → `User` → `Time` → $\mathbb{Q}^{\geq 0}$: defines a mapping which stores the z-balances value for a token and a user at a given point in time.
- $z\_allowances$: `Token` → `User` → `User` → $\mathbb{Q}^{\geq 0}$: stores the z-allowance value for a given user and token.
- $debts$: `Token` → `User` → `Time` → $\mathbb{Q}^{\geq 0}$: defines a mapping that stores the debt of a user given a token at a given point in time.
- $collateral$: `User` → $\mathcal{P}$(`Token`): stores the set of tokens that can be used as collateral for a given user.
- $owner$: `User` ∪ $\{none\}$: current owner. The value $none$ denotes that no $owner$ was defined by operation $set\_ownership$.
- $treasury$: `User`: current treasury value.
- $enabled\_tokens$: $\mathcal{P}$(`Token`): set of tokens enabled in the protocol state.
- $borrow\_factor$: `Token` → $[0, 1]$: current borrow factor for a given token.
- $collateral\_factor$: `Token` → $[0, 1]$: current collateral factor for a given token.
- $flash\_loan\_fee$: `Token` → $\mathbb{Q}^{\geq 0}$: current fee for flash loans for a given token.
- $liquidation\_bonus$: `Token` → $\mathbb{Q}^{\geq 0}$: current liquidation bonus for a given token.
- $reserve\_factor$: `Token` → $[0, 1]$: current reserve factor for a given token.

The Coq implementation for state type is available online at:
https://github.com/NethermindEth/zklend-protocol-fv/blob/203bcbf1759dff9d80fb5f69837a3d4d89c9cb67/specs/state/abstract_state.v#L8.

4.1.2 State Invariants

The protocol `State` definition is subject to the following restrictions on debts and z-balances:

$$\frac{\partial debts}{\partial time} = borrowing\_rate(state, token, time) \times debts(token, user, time)$$

$$\frac{\partial z\_balances}{\partial time} = \begin{cases} utilisation\_rate(state, token, time) \times \\ borrowing\_rate(state, token, time) \times \\ z\_balances(token, user, time) \times \\ (1 - reserve\_factor(token)) & , user \neq treasury \\ \\ utilisation\_rate(state, token, time) \times \\ borrowing\_rate(state, token, time) \times \\ z\_balances(token, user, time) \times \\ (1 - reserve\_factor(token)) + \\ \sum\limits_{u \in \mathtt{User}} \begin{pmatrix} utilisation\_rate(state, token, time) \times \\ borrowing\_rate(state, token, time) \times \\ z\_balances(token, u, time) \times \\ reserve\_factor(token) \end{pmatrix} & , user = treasury \end{cases}$$

4.1.3 Model parameters

Our model uses some parameters that cannot be expressed as part of the protocol's state. These parameters denote external factors like the current US dollar value for a specific token or the value of the interest rate for a token. Both parameters are represented by the following maps:

- $usd\_value \in \mathtt{Token} \to \mathtt{Time} \to \mathbb{Q}^{\geq 0}$: the price in USD of each token at a given point in time;
- $borrowing\_rate \in\!\!\to \mathtt{Token} \to \mathtt{Time} \to \mathbb{Q}^{\geq 0}$: interest rate on borrowings for a given token.

The encoding of these parameters are available at the following link.

4.1.4 Helper functions

We need to define some auxiliary functions to specify the relevant protocol properties. The first one, $collateral\_value$, computes the total US dollar value used as collateral by a user in a given instant of time. The second one computes the utilization rate for a given token at a point in time.

$$collateral\_value(\mathtt{state}, user, time) = \sum_{t \in \mathtt{state}.collateral(user)} \begin{pmatrix} \mathtt{state}.z\_balances(t, user, time) \times \\ usd\_value(t, time) \times \\ \mathtt{state}.collateral\_factor(t) \end{pmatrix}$$

$$utilisation\_rate(\mathtt{state}, token, time) = \frac{\sum\limits_{u \in \mathtt{User}} \mathtt{state}.debts(token, u, time)}{\begin{pmatrix} \mathtt{state}.erc20\_balances(token, market) + \\ \sum\limits_{u \in \mathtt{User}} \mathtt{state}.debts(token, u, time) \end{pmatrix}}$$

The function $collateral\_value$ definition can be found here, and the definition for $utilisation\_rate$ is available here.

4.2 Invariants

In this section, we describe important invariants that are used to specify the correctness of the ZkLend protocol: the state well-formedness and solvency. The first property specifies when the protocol state is considered valid, and the second ensures that it is always possible for the protocol to recover from an insolvent state using a liquidation operation. Section 4.2.1 describes the well-formedness predicate over the protocol state, and Section 4.2.2 defines the solvency condition predicate.

4.2.1 State well-formedness

We define a well-formed state as one with all its members defined as total functions over its domains. This property is needed to ensure the correct behavior of all protocol operations. For each state component, we have a property for its well-formedness. As an example, we define that the $enabled\_tokens$ state member is well-formed if it is a subset of all tokens available at the state. This property is represented by the following Coq definition:

```
Definition wf_enabled_tokens (s : State) : Prop :=
  list_subset (enabled_tokens s) (tokens s).
```

The function `list_subset` holds when the first argument is a subset of the second. The functions `enabled_tokens` and `tokens` return the set of enabled tokens and all tokens defined in the state `s`. Most of the state members involve finite mappings between tokens/users and monetary values. The Well-formedness of such mappings is defined by universally quantifying them over all possible user/token values available at the protocol state. Details about these definitions can be found here.

Using different properties for each state member allows us to express its well-formedness as the conjunction of the well-formedness of its members as follows:

```
Definition wf_state (s : State)(tm : Time) : Prop :=
  wf_erc20_balances s   ∧
  wf_erc20_allowances s ∧
  wf_z_balances s tm    ∧
  wf_z_allowances s     ∧
  wf_debts s tm         ∧
  wf_collateral s       ∧
  wf_owner s            ∧
  wf_treasury s         ∧
  wf_enabled_tokens s   ∧
  wf_borrow_factor s    ∧
  wf_collateral_factor s ∧
  wf_liquidation_bonus s ∧
  wf_reserve_factor s   ∧
  In market (users s).
```

Definition `wf_state` defines that a well-formed state is one that has only well-formed components and has a `market` value defined in the set of all protocol users.

4.2.2 Solvency predicate

An important property of a financial market protocol, like ZkLend, is solvency. Intuitively, the solvency property guarantees that the protocol will be able to meet its debts. We define solvency as two sub-properties. The first ensures that for all available tokens, the total sum of all user's z-balances for a given token is less than the sum of the market's balance for that token with all users' debts for this same token. Formally:

$$\forall tok \in \texttt{Token}, t \in \texttt{Time}, \sum_{u \in \texttt{User}} s.z\_balances(tok, u, t) \leq s.erc20\_balances(tok, market) + \sum_{u \in \texttt{User}} s.debts(tok, u, t)$$

The Coq implementation for this component of the solvency property can be found here.

The second component of solvency property states that any protocol user either has enough collateral to cover its debts in USD or it can be liquidated by other user in order to reduce its debts. Formally:

$$\forall u \in \texttt{User}, t \in \texttt{Time}. \left(\begin{array}{r} \left(\sum_{tok \in \texttt{Token}} (s.debts(tok, u, t) \times usd\_value(tok, t)) \leq collateral\_value(s, u, t) \right) \vee \\ \left(\begin{array}{r} \exists s' \in , u' \in \texttt{User}, dt \in \texttt{Token}, \quad \forall d \in \mathbb{Q}^{>0}, \quad s.debts(dt, u, t) > d \quad \Rightarrow \\ s \xrightarrow{\texttt{liquidate}(u, dt, d, \_), u', t} s' \quad \wedge \\ \sum_{tok \in \texttt{Token}} (s.debts(tok, u, t) \times usd\_value(tok, t)) - collateral\_value(s, u, t) \quad > \\ \sum_{tok \in \texttt{Token}} (s'.debts(tok, u, t) \times usd\_value(tok, t)) - collateral\_value(s', u, t) \end{array} \right) \end{array} \right)$$

The Coq implementation of the second component of the solvent property can be found here. Finally, we say that the protocol state is solvent if it satisfies both of the previously presented properties.

4.3 Operation Specifications

This section presents the specification of each operation of the ZkLend protocol. Specifications are written as a state transition system using the following notation:

$$\frac{\begin{array}{c} condition_1 \\ \vdots \\ condition_n \end{array}}{old\_state \xrightarrow{\texttt{operation}(arguments), \#caller, \#block\_timestamp} new\_state}$$

This example specification states that in an old_state, if all $conditions_i$ holds, then a call, from $\#caller$ and $\#block\_timestamp$, to the operation `op` with arguments $args$, produces a new_state. In the next sections, we present the specifications for both `Market` and `ZToken` contracts.

4.4 Specification of Market's Functions

4.4.1 Specification of the `deposit` function (Coq code)

The `deposit` function accepts a token and an amount and modifies the associated balances/allowances within the protocol state. To execute correctly, the function requires the following conditions to be met: (1) the caller possesses sufficient balance for the deposit, and (2) the deposit function only alters the caller's balance.

$$\frac{\begin{array}{l} amount \in \mathbb{Q}^{>0} \\ \#caller \in \texttt{User} \\ token \in enabled\_tokens \\ old\_z\_balances, new\_z\_balances \in \texttt{User} \to \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ old\_user\_erc20\_balance, new\_user\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\ old\_reserve\_erc20\_balance, new\_reserve\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\ old\_erc20\_allowance, new\_erc20\_allowance \in \mathbb{Q}^{\geq 0} \\ \\ old\_erc20\_allowance \geq amount \\ old\_user\_erc20\_balance \geq amount \\ \\ new\_erc20\_allowance = old\_erc20\_allowance - amount \\ new\_reserve\_erc20\_balance = old\_reserve\_erc20\_balance + amount \\ new\_user\_erc20\_balance = old\_user\_erc20\_balance - amount \\ new\_z\_balances(\#caller, \#block\_timestamp) = old\_z\_balances(\#caller, \#block\_timestamp) + amount \\ \forall u \in \texttt{User} \setminus \{\#caller\}. new\_z\_balances(u, \#block\_timestamp) = old\_z\_balances(u, \#block\_timestamp) \end{array}}{state \left\{ \begin{array}{c} z\_balances(token) = old\_z\_balances, \\ erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance, \\ erc20\_allowances(token, \#caller, market) = old\_erc20\_allowance \end{array} \right\} \xrightarrow{\texttt{deposit}(token, amount), \#caller, \#block\_timestamp} state \left\{ \begin{array}{c} z\_balances(token) = new\_z\_balances, \\ erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance, \\ erc20\_allowances(token, \#caller, market) = new\_erc20\_allowance \end{array} \right\}$$

Figure 1: Specification for the `deposit` function.

4.4.2 Specification of the `withdraw` function (Coq code)

The `withdraw` function takes in a token and an amount and adjusts the related balances/allowances within the protocol state. For the function to execute correctly, the following conditions must be met: (1) adequate balances must be present in the market for the withdrawal, (2) solely the caller's balance is affected by the function, and (3) the caller's address must possess enough collateral value in the current protocol state.

$$\frac{\begin{array}{l} amount \in \mathbb{Q}^{>0} \\ \#caller \in \texttt{User} \\ token \in enabled\_tokens \\ old\_z\_balances, new\_z\_balances \in \texttt{User} \to \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ old\_user\_erc20\_balance, new\_user\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\ old\_reserve\_erc20\_balance, new\_reserve\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\ \\ old\_z\_balances \geq amount \\ old\_reserve\_erc20\_balance \geq amount \\ \\ new\_reserve\_erc20\_balance = old\_reserve\_erc20\_balance - amount \\ new\_user\_erc20\_balance = old\_user\_erc20\_balance + amount \\ new\_z\_balances(\#caller, \#block\_timestamp) = old\_z\_balances(\#caller, \#block\_timestamp) - amount \\ \forall u \in \texttt{User} \setminus \{\#caller\}. new\_z\_balances(u, \#block\_timestamp) = old\_z\_balances(u, \#block\_timestamp) \\ \\ \sum_{t \in \texttt{Token}} \left(\frac{debts(t, \#caller, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, \#caller, \#block\_timestamp) \end{array}}{state \left\{ \begin{array}{c} z\_balances(token) = old\_z\_balances, \\ erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance \end{array} \right\} \xrightarrow{\texttt{withdraw}(token, amount), \#caller, \#block\_timestamp} s@state \left\{ \begin{array}{c} z\_balances(token) = new\_z\_balances, \\ erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance \end{array} \right\}$$

Figure 2: Specification for the `withdraw` function.

4.4.3 Specification for the `withdraw_all` function (Coq code)

The `withdraw_all` function transfers the caller's entire z-balance from the market and adds it to their current ERC20 balance. In essence, it follows the same pre-conditions as withdrawal and resets the caller's z-balance to zero in the resulting state.

$$
\begin{array}{l}
\#caller \in \texttt{User} \\
token \in enabled\_tokens \\
old\_z\_balances, new\_z\_balances \in \texttt{User} \to \texttt{Time} \to \mathbb{Q}^{\geq 0} \\
old\_user\_erc20\_balance, new\_user\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\
old\_reserve\_erc20\_balance, new\_reserve\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\
\\
old\_reserve\_erc20\_balance \geq old\_z\_balances(\#caller, \#block\_timestamp) \\
\\
new\_reserve\_erc20\_balance = old\_reserve\_erc20\_balance - old\_z\_balances(\#caller, \#block\_timestamp) \\
new\_user\_erc20\_balance = old\_user\_erc20\_balance + old\_z\_balances(\#caller, \#block\_timestamp) \\
new\_z\_balances(\#caller, \#block\_timestamp) = 0 \\
\forall u \in \texttt{User} \setminus \{\#caller\}. new\_z\_balances(u, \#block\_timestamp) = old\_z\_balances(u, \#block\_timestamp) \\
\\
\dfrac{\sum\limits_{t \in \texttt{Token}} \left(\frac{debts(t, \#caller, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, \#caller, \#block\_timestamp)}{state \left\{ \begin{array}{c} z\_balances(token) = old\_z\_balances, \\ erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance \end{array} \right\} \xrightarrow{\texttt{withdraw\_all}(token), \#caller, \#block\_timestamp} s@state \left\{ \begin{array}{c} z\_balances(token) = new\_z\_balances, \\ erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance \end{array} \right\}}
\end{array}
$$

Figure 3: Specification for the `withdraw_all` functions.

4.4.4 Specification for the `borrow` function (Coq code)

The `borrow` function accepts a token and an amount as inputs and utilizes the token's market reserves to augment the caller's token balance by the specified amount. For the function to run without error, the following prerequisites must be met: (1) sufficient market reserves are available for borrowing, (2) the caller has adequate collateral value, and (3) the borrow function solely modifies the outstanding debts for the caller and the specified token.

$$
\begin{array}{l}
amount \in \mathbb{Q}^{>0} \\
\#caller \in \texttt{User} \\
token \in enabled\_tokens \\
old\_debts, new\_debts \in \texttt{Token} \to \texttt{User} \to \texttt{Time} \to \mathbb{Q}^{\geq 0} \\
old\_reserve\_erc20\_balance, new\_reserve\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\
old\_user\_erc20\_balance, new\_user\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\
\\
old\_reserve\_erc20\_balance \geq amount \\
\\
new\_reserve\_erc20\_balance = old\_reserve\_erc20\_balance - amount \\
new\_user\_erc20\_balance = old\_user\_erc20\_balance + amount \\
new\_debts(token, \#caller, \#block\_timestamp) = old\_debts(token, \#caller, \#block\_timestamp) + amount \\
\forall (t, u) \in \texttt{Token} \times \texttt{User} \setminus \{(token, \#caller)\}. new\_debts(t, u, \#block\_timestamp) = old\_debts(t, u, \#block\_timestamp) \\
\\
\dfrac{\sum\limits_{t \in \texttt{Token}} \left(\frac{new\_debts(t, \#caller, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, \#caller, \#block\_timestamp)}{state \left\{ \begin{array}{c} erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance, \\ debts = old\_debts \end{array} \right\} \xrightarrow{\texttt{borrow}(token, amount), \#caller, \#block\_timestamp} s@state \left\{ \begin{array}{c} erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance, \\ debts = new\_debts \end{array} \right\}}
\end{array}
$$

Figure 4: Specification of the `borrow` function

4.4.5 Specification of `repay` function (Coq code)

The `repay` function accepts a token value and an amount, which allows the caller to use the specified token to pay off a portion of their debts with the market. In order for the `repay` function to run successfully, two conditions must be met: (1) the input token must be enabled in the protocol, and (2) the caller must have enough balance to pay the specified amount to the market.

$$\frac{\begin{array}{l} amount \in \mathbb{Q}^{>0} \\ \#caller \in \texttt{User} \\ token \in enabled\_tokens \\ old\_reserve\_erc20\_balance, new\_reserve\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\ old\_user\_erc20\_balance, new\_user\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\ old\_user\_debt, new\_user\_debt \in \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ \\ old\_user\_debt(\#block\_timestamp) \geq amount \\ old\_user\_erc20\_balance \geq amount \\ \\ new\_user\_debt(\#block\_timestamp) = old\_user\_debt(\#block\_timestamp) - amount \\ new\_user\_erc20\_balance = old\_user\_erc20\_balance - amount \\ new\_reserve\_erc20\_balance = old\_reserve\_erc20\_balance + amount \end{array}}{\begin{array}{c} state\left\{\begin{array}{c} erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance \\ debts(token, \#caller) = old\_user\_debt, \end{array}\right\} \\ \xrightarrow{\texttt{repay}(token, amount), \#caller, \#block\_timestamp} \\ state\left\{\begin{array}{c} erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance \\ debts(token, \#caller) = new\_user\_debt \end{array}\right\} \end{array}}$$

Figure 5: Specification of `repay` function.

4.4.6 Specification of `repay_all` function (Coq code)

The `repay_all` function allows the caller to use a specified token value to pay off all their debts with the market. For the function to work properly, two conditions must be met: (1) the protocol must have enabled the input token, and (2) the caller must have a sufficient balance to fully pay off their debts to the market.

$$\frac{\begin{array}{l} \#caller \in \texttt{User} \\ token \in enabled\_tokens \\ old\_user\_debt \in \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ new\_user\_debt \in \texttt{Time} \to 0 \\ old\_user\_erc20\_balance, new\_user\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\ old\_reserve\_erc20\_balance, new\_reserve\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\ \\ old\_user\_erc20\_balance \geq old\_user\_debt(\#block\_timestamp) \\ \\ new\_user\_erc20\_balance = old\_user\_erc20\_balance - old\_user\_debt(\#block\_timestamp) \\ new\_reserve\_erc20\_balance = old\_reserve\_erc20\_balance + old\_user\_debt(\#block\_timestamp) \end{array}}{\begin{array}{c} state\left\{\begin{array}{c} erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance, \\ debts(token, \#caller) = old\_user\_debt \end{array}\right\} \\ \xrightarrow{\texttt{repay\_all}(token), \#caller, \#block\_timestamp} \\ state\left\{\begin{array}{c} erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance, \\ debts(token, \#caller) = 0 \end{array}\right\} \end{array}}$$

Figure 6: Specification of `repay_all` function

4.4.7 Specification for the `liquidate` function (Coq code)

Once the liquidation threshold is reached, any protocol user can invoke the liquidate operation, provided they possess the appropriate collateral type and enough quantity to cover the borrower's loan position. Liquidators can utilize the liquidation contract repeatedly until the borrower's loan position returns to its borrowing capacity. This guarantees a way to manage protocol risk. The function liquidate can be successfully executed when the following requirements are met: (1) the total USD value of the user's debts is greater or equal to the USD value of his collateral; (2) the user has enough z-balances to cover the collateral amount calculated using the input debt amount and the current USD price.

$$
\begin{array}{l}
\#caller \in \texttt{User} \\
collateral\_amount \in \mathbb{Q}^{>0} \\
collateral\_token \in collateral(user) \\
debt\_amount \in \mathbb{Q}^{>0} \\
debt\_token \in enabled\_tokens \\
user \in \texttt{User} \\
old\_debts, new\_debts \in \texttt{Token} \to \texttt{User} \to \texttt{Time} \to \mathbb{Q}^{\geq 0} \\
old\_reserve\_erc20\_balance, new\_reserve\_erc20\_balance \in \mathbb{Q}^{\geq 0} \\
old\_caller\_erc20\_balance, new\_caller\_erc20\_balance \in \mathbb{Q}^{\geq 0}
\end{array}
$$

$$
\begin{array}{l}
collateral\_amount = debt\_amount \times \dfrac{usd\_price(debt\_token, \#block\_timestamp)}{usd\_price(collateral\_token, \#block\_timestamp)} \times (1 + liquidation\_bonus(collateral\_token)) \\
old\_debts(debt\_token, user, \#block\_timestamp) \geq debt\_amount \\
old\_caller\_erc20\_balance \geq debt\_amount \\
old\_z\_balances(user, \#block\_timestamp) \geq collateral\_amount
\end{array}
$$

$$
\begin{array}{l}
new\_debts(debt\_token, user, \#block\_timestamp) = old\_debts(debt\_token, user, \#block\_timestamp) - debt\_amount \\
\forall (t, u) \in \texttt{Token} \times \texttt{User} \setminus \{(debt\_token, user)\}. new\_debts(t, u, \#block\_timestamp) = old\_debts(t, u, \#block\_timestamp) \\
new\_caller\_erc20\_balance = old\_caller\_erc20\_balance - debt\_amount \\
new\_reserve\_erc20\_balance = old\_reserve\_erc20\_balance + debt\_amount \\
new\_z\_balances(user, \#block\_timestamp) = old\_z\_balances(user, \#block\_timestamp) - collateral\_amount \\
new\_z\_balances(\#caller, \#block\_timestamp) = old\_z\_balances(\#caller, \#block\_timestamp) + collateral\_amount \\
\forall u \in \texttt{User} \setminus \{\#caller, user\}. new\_z\_balances(u, \#block\_timestamp) = old\_z\_balances(u, \#block\_timestamp)
\end{array}
$$

$$
\frac{\sum_{t \in \texttt{Token}} (new\_debts(t, user, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)) \geq collateral\_value(s, user, \#block\_timestamp)}{
state \left\{ \begin{array}{c} erc20\_balances(debt\_token, \#caller) = old\_caller\_erc20\_balance, \\ erc20\_balances(debt\_token, market) = old\_reserve\_erc20\_balance, \\ debts = old\_debts, \\ z\_balances(collateral\_token) = old\_z\_balances \end{array} \right\}
\xrightarrow{\texttt{liquidate}(user, debt\_token, debt\_amount, collateral\_token), \#caller, \#block\_timestamp}
s@state \left\{ \begin{array}{c} erc20\_balances(debt\_token, \#caller) = new\_caller\_erc20\_balance, \\ erc20\_balances(debt\_token, market) = new\_reserve\_erc20\_balance, \\ debts = new\_debts, \\ z\_balances(collateral\_token) = new\_z\_balances \end{array} \right\}
}
$$

Figure 7: Specification for the `liquidate` function.

4.4.8 Specification of the function `enable_collateral` (Coq code)

The function `enable_collateral` marks that a specific token can be used as collateral.

$$
\frac{\begin{array}{c} \#caller \in \texttt{User} \\ token \in enabled\_tokens \setminus old\_collateral \end{array}}{
state \left\{ \; collateral(\#caller) = old\_collateral \; \right\}
\xrightarrow{\texttt{enable\_collateral}(token), \#caller}
state \left\{ \; collateral(\#caller) = old\_collateral \cup \{token\} \; \right\}
}
$$

Figure 8: Specification of the function `enable_collateral`

4.4.9 Specification of the function `disable_collateral` (Coq code)

The function `disable_collateral` allows removing some tokens from the set of tokens allowed to be used as collateral. The function `disable_collateral` requires the caller's debts to be less than his collateral value.

$$\frac{\begin{array}{l}\#caller \in \texttt{User} \\ token \in old\_collateral \\ \sum_{t \in \texttt{Token}} \left(\frac{debts(t, \#caller, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, \#caller, \#block\_timestamp)\end{array}}{\begin{array}{c} state\left\{ \ collateral(\#caller) = old\_collateral \ \right\} \\ \xrightarrow{\texttt{disable\_collateral}(token), \#caller} \\ s@state\left\{ \ collateral(\#caller) = old\_collateral \setminus \{token\} \ \right\}\end{array}}$$

Figure 9: Specifications for the `disable_collateral` functions.

4.4.10 Specification of the permissioned entry points (Coq code)

All functions listed here are only callable by the owner. The `add_reserve` functions add a new token, not in the set of enabled tokens, to the set of enabled tokens. The functions `set_treasury` sets a new treasury. The `transfer_ownership` and `renounce_ownership` deal with protocol ownership. `transfer_ownership` changes the owner and `renounce_ownership` leaves the owner position empty, setting it to $none$.

$$\frac{\begin{array}{l}\#caller \in \texttt{User} \\ token \in \texttt{Token} \setminus old\_enabled\_tokens \\ \\ \#caller = owner\end{array}}{\begin{array}{c} state\left\{ \ enabled\_tokens = old\_enabled\_tokens \ \right\} \\ \xrightarrow{\texttt{add\_reserve}(token, z\_token)} \\ state\left\{ \ enabled\_tokens = old\_enabled\_tokens \cup \{token\} \ \right\}\end{array}}$$

$$\frac{\begin{array}{l}\#caller \in \texttt{User} \\ old\_treasury, new\_treasury \in \texttt{User} \\ \\ \#caller = owner\end{array}}{\begin{array}{c} state\left\{ \ treasury = old\_treasury \ \right\} \\ \xrightarrow{\texttt{set\_treasury}(new\_treasury), \#caller} \\ state\left\{ \ treasury = new\_treasury \ \right\}\end{array}}$$

$$\frac{\begin{array}{l} old\_owner, new\_owner \in \texttt{User} \\ \\ \#caller = old\_owner\end{array}}{\begin{array}{c} state\left\{ \ owner = old\_owner \ \right\} \\ \xrightarrow{\texttt{transfer\_ownership}(new\_owner), \#caller} \\ state\left\{ \ owner = new\_owner \ \right\}\end{array}}$$

$$\frac{\begin{array}{l} old\_owner \in \texttt{User} \\ \\ \#caller = old\_owner\end{array}}{\begin{array}{c} state\left\{ \ owner = old\_owner \ \right\} \\ \xrightarrow{\texttt{renounce\_ownership}(), \#caller} \\ state\left\{ \ owner = none \ \right\}\end{array}}$$

Figure 10: Specification for the permissioned entry points: `add_reserve`, `set_treasury`, `transfer_ownership`, and `renounce_ownership`.

4.5 Specification of ZToken Functions

4.5.1 Specification of the `transfer` function (Coq code)

Transfer a token amount to the recipient, with the condition that the caller must possess enough collateral value in the current protocol state. If the caller and the recipient are the same, the state doesn't change.

$$\frac{\begin{array}{l} amount \in \mathbb{Q}^{>0} \\ \#caller \in \texttt{User} \\ old\_from\_balance, new\_from\_balance \in \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ old\_to\_balance, new\_to\_balance \in \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ recipient \in \texttt{User} \setminus \{\#caller\} \\ token \in \texttt{Token} \\ \\ old\_from\_balance(\#block\_timestamp) \geq amount \\ \\ new\_from\_balance(\#block\_timestamp) = old\_from\_balance(\#block\_timestamp) - amount \\ new\_to\_balance(\#block\_timestamp) = old\_to\_balance(\#block\_timestamp) + amount \\ \\ \sum_{t \in \texttt{Token}} \left(\frac{debts(t, \#caller, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, \#caller, \#block\_timestamp) \end{array}}{\begin{array}{c} state \left\{ \begin{array}{c} z\_balances(token, \#caller) = old\_from\_balance, \\ z\_balances(token, recipient) = old\_to\_balance \end{array} \right\} \\ \xrightarrow{\texttt{transfer}(recipient, amount), token, \#caller, \#block\_timestamp} \\ s@state \left\{ \begin{array}{c} z\_balances(token, \#caller) = new\_from\_balance, \\ z\_balances(token, recipient) = new\_to\_balance \end{array} \right\} \end{array}}$$

$$\frac{\begin{array}{l} amount \in \mathbb{Q}^{>0} \\ \#caller \in \texttt{User} \\ token \in \texttt{Token} \\ \\ recipient = \#caller \\ z\_balances(token, \#caller, \#block\_timestamp) \geq amount \\ \\ \sum_{t \in \texttt{Token}} \left(\frac{debts(t, \#caller, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, \#caller, \#block\_timestamp) \end{array}}{state \xrightarrow{\texttt{transfer}(recipient, amount), token, \#caller, \#block\_timestamp} state}$$

Figure 11: Specifications for the `transfer` function.

4.5.2 Specification of the `transfer_all` function (Coq code)

The function `transfer_all` has the same behavior of `transfer` function, but it transfers all the z-balance from the caller to the recipient, with the condition that the caller must possess enough collateral value in the current protocol state.

$$\frac{\begin{array}{l}\#caller \in \texttt{User} \\ new\_from\_balance, old\_from\_balance \in \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ new\_to\_balance, old\_to\_balance \in \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ recipient \in \texttt{User} \setminus \{\#caller\} \\ token \in \texttt{Token} \\ \\ new\_from\_balance(\#block\_timestamp) = 0 \\ new\_to\_balance(\#block\_timestamp) = old\_to\_balance(\#block\_timestamp) + old\_from\_balance(\#block\_timestamp) \\ \\ \sum_{t \in \texttt{Token}} \left(\frac{debts(t, \#caller, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, \#caller, \#block\_timestamp)\end{array}}{\begin{array}{c} state \left\{ \begin{array}{c} z\_balances(token, \#caller) = old\_from\_balance, \\ z\_balances(token, recipient) = old\_to\_balance \end{array} \right\} \\ \xrightarrow{\texttt{transfer\_all}(recipient), token, \#caller, \#block\_timestamp} \\ s@state \left\{ \begin{array}{c} z\_balances(token, market) = new\_from\_balance, \\ z\_balances(token, recipient) = new\_to\_balance \end{array} \right\} \end{array}}$$

$$\frac{\begin{array}{l}\#caller \in \texttt{User} \\ token \in \texttt{Token} \\ \\ recipient = \#caller \\ z\_balances(token, \#caller, \#block\_timestamp) > 0 \\ \\ \sum_{t \in \texttt{Token}} \left(\frac{debts(t, \#caller, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, \#caller, \#block\_timestamp)\end{array}}{state \xrightarrow{\texttt{transfer\_all}(recipient), token, \#caller, \#block\_timestamp} state}$$

Figure 12: Specifications for the `transfer_all` function.

4.5.3 Specification of the `transferFrom` function (Coq code)

Transfer a number of tokens to the recipient from the sender, with the condition that the sender must possess enough collateral value in the current protocol state. If the recipient and the sender are the same, update the allowances accordingly.

$$\frac{\begin{array}{l} amount \in \mathbb{Q}^{>0} \\ old\_allowance, new\_allowance \in \mathbb{Q}^{\geq 0} \\ old\_from\_balance, new\_from\_balance \in \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ old\_to\_balance, new\_to\_balance \in \texttt{Time} \to \mathbb{Q}^{\geq 0} \\ sender \in \texttt{User} \\ recipient \in \texttt{User} \setminus \{sender\} \\ token \in \texttt{Token} \\ \\ old\_allowance \geq amount \\ old\_from\_balance(\#block\_timestamp) \geq amount \\ \\ new\_allowance = old\_allowance - amount \\ new\_from\_balance(\#block\_timestamp) = old\_from\_balance(\#block\_timestamp) - amount \\ new\_to\_balance(\#block\_timestamp) = old\_to\_balance(\#block\_timestamp) + amount \\ \\ \sum_{t \in \texttt{Token}} \left(\frac{debts(t, sender, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, sender, \#block\_timestamp) \end{array}}{\begin{array}{c} state \left\{ \begin{array}{c} z\_allowances(token, sender, \#caller) = old\_allowance, \\ z\_balances(token, sender) = old\_from\_balance, \\ z\_balances(token, recipient) = old\_to\_balance \end{array} \right\} \\ \xrightarrow{\texttt{transferFrom}(sender, recipient, amount), token, \#caller, \#block\_timestamp} \\ s@state \left\{ \begin{array}{c} z\_allowances(token, sender, \#caller) = new\_allowance, \\ z\_balances(token, sender) = new\_from\_balance, \\ z\_balances(token, recipient) = new\_to\_balance \end{array} \right\} \end{array}}$$

$$\frac{\begin{array}{l} amount \in \mathbb{Q}^{>0} \\ old\_allowance, new\_allowance \in \mathbb{Q}^{\geq 0} \\ sender \in \texttt{User} \\ token \in \texttt{Token} \\ \\ recipient = sender \\ new\_allowance = old\_allowance - amount \\ \\ old\_allowance \geq amount \\ z\_balances(token, sender, \#block\_timestamp) \geq amount \\ \\ \sum_{t \in \texttt{Token}} \left(\frac{debts(t, sender, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, sender, \#block\_timestamp) \end{array}}{\begin{array}{c} state \left\{ \; z\_allowances(token, sender, \#caller) = old\_allowance, \; \right\} \\ \xrightarrow{\texttt{transferFrom}(sender, recipient, amount), token, \#caller, \#block\_timestamp} \\ s@state \left\{ \; z\_allowances(token, sender, \#caller) = new\_allowance, \; \right\} \end{array}}$$

Figure 13: Specifications for the `transferFrom` functions.

4.5.4 Specification of the `approve` function (Coq code)

The `approve` function updates the allowance of a spender to a certain amount.

$$\frac{\begin{array}{l} amount \in \mathbb{Q}^{\geq 0} \\ \#caller, spender \in \texttt{User} \\ token \in \texttt{Token} \end{array}}{state \xrightarrow{\texttt{approve}(spender, amount), token, \#caller} state \{z\_allowances(\#caller, spender) = amount\}}$$

Figure 14: Specification for the `approve` function.

4.6 Specification correctness

In this section, we discuss the correctness of the specification of the ZkLend protocol. We used the Coq proof assistant to define the function which translates concrete states into abstract states, available here, and prove it preserves well-formedness, as stated by the following theorem:

```
Theorem abstract_fun_preserve_wf:
  forall cs tm, wf_concrete_state cs tm → wf_state (concrete_to_abstract cs tm) tm.
```

The complete formalization of the preservation of well-formedness is available here. We also defined each protocol function specification and then proved that such functions satisfy the invariants of the first solvency condition. As an example, we present only two of such solvency condition proofs below: *repay* from Market and *transfer* from ZToken.

```
Lemma repay_preserves_solvent_condition1
  : forall s m tm,
      wf_state s tm →
      state_solvent s m →
    forall u1 t1 tm1 v,
      In u1 (users s) →
      In t1 (enabled_tokens s) →
      0 < v →
      solvent_condition1 (repay t1 v u1 tm1 m s).
```

```
Lemma transfer_preserves_solvent_condition1
  : forall s tm m,
      wf_state s tm →
      state_solvent s m →
    forall t v u1 u2, In t (tokens s) →
      In u1 (users s) →
      In u2 (users s) →
      solvent_condition1 (transfer t v u1 u2 tm m s).
```

The next table presents the links for the solvency condition 1 proofs for market contract operations.

Protocol Operation	Link
Market.add_reserve	add_reserve solvency condition 1 proof.
Market.disable_collateral	disable_collateral solvency condition 1 proof.
Market.enable_collateral	enable_collateral solvency condition 1 proof.
Market.renounce_ownership	renouce_ownership solvency condition 1 proof.
Market.repay	repay solvency condition 1 proof.
Market.set_treasury	set_treasury solvency condition 1 proof.
Market.transfer_ownership	transfer_ownership solvency condition 1 proof.

Table 2: Solvency condition 1 proofs for Market contract operations

Links for the solvency condition 1 proofs ZToken contract operations are presented in the next table.

Protocol Operation	Link
ZToken.approve	approve solvency condition 1 proof.
ZToken.transfer	transfer solvency condition 1 proof.
ZToken.transfer_all	transfer_all solvency condition 1 proof.
ZToken.transfer_from	transfer_from solvency condition 1 proof.

Table 3: Solvency condition 1 proof for ZToken contract operations

4.7 Limitations

Note that this model has some limitations with respect to modeling implementation of the ZkLend money-market protocol: a) we model token amounts using rational numbers, $\mathbb{Q}$, when the implementation, in fact, uses a fixed point representation. Due to this, the model eschews issues relating to rounding; b) we model the $debts$ and $z\_balances$ as being discretely compounded in the implementation, but in the ZkLend whitepaper, these are modeled by a continuous curve.

4.8 Final Remarks

This report has introduced a formalized model of the ZkLend protocol in the Coq proof assistant. It introduces an abstract model of the protocol's state. In the repository, one can also find the definition of the ZkLend protocol implementation's concrete state and a map from concrete to abstract states. We have identified and defined the key invariants that underlay the protocol's security and introduced abstract specifications for the operations of the Market and ZToken contracts and formalized them. Finally, we have introduced formalized proofs that our abstract specification satisfies the well-formedness invariant and the first conjunct of the solvency invariant. These specifications and the mapping from concrete to abstract states can now be used to develop a property tester for the ZkLend protocol using QuickChick[1].

References

- [1] Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Ed. by Benjamin C. Pierce. Vol. 4. Software Foundations. Version 1.3.2, http://softwarefoundations.cis.upenn.edu. Electronic textbook, 2023.

  The coq proof assistant. URL: https://coq.inria.fr/.

  zkLend whitepaper. URL: https://zklend.com/documents/2022.12.02_zkLend-White-Paper.pdf.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

[2] [3]

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited: Data/state integrity, loss of availability, financial loss, reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] Flash Loan fees do not increase the ZToken accumulator value

File(s): internals/Market/storage.cairo

Description: On a successful flash loan the fees paid by the loan recipient are sent to the market contract but the ZToken's lending accumulator value does not increase proportionately alongside the fees.

Upon a successful flash loan, the borrower pays their fees and the two functions `update_accumulators(...)` and `update_rates_and_raw_total_debt(...)`. The function `update_accumulators(...)` does no balance checks so it cannot account for flash loan fees, and `update_rates_and_raw_total_debt(...)` will calculate a new lending/borrowing rate with a slightly lower liquidity utilization rate due to the extra fees in the reserve. None of these functions change the ZToken accumulator to take the flash loan fees into account. These flash loan fees are permanently lost in the pool and cannot be recovered.

Recommendation(s): Consider measuring the number of flash loan fees paid and proportionately increase the ZToken accumulator so that the value of the ZTokens correctly represents the number of assets in the reserve.

Status: Fixed

Update from the client: A fix has been implemented in PR /33, which also adds a new test case `test_flashloan_fee_distribution` for validating the fix.

6.2 [Medium] Application must be robust and protect itself against oracle malfunction

File(s): zklend/*

Description: The application depends on the external oracle Empiric Network. According to the Empiric documentation, the oracle has not been audited at the time of this audit. The external oracle provides the price of assets for ZkLend. Having a robust application for dealing with external entities malfunctioning is very important to protect the users' assets. The application must be able to identify and promptly respond to unexpected events originating in external applications. In this sense, some questions need to be addressed by the application, such as: a) What happens if the external oracle is not responding? b) What happens if the external oracle is hacked? No matter what happens, users' assets must be preserved and no malicious trades can happen in the application.

Recommendation(s): Consider strategies to protect the application against Oracle malfunctions.

Status: Acknowledged

Update from the client: We agree that it's important to protect against oracle malfunction/exploits, and a simple staleness check won't suffice. We plan to implement Oracle protection with a sanity range check with prices from layer-1 Uniswap pools to make sure out-of-range prices reported from the Oracle network are rejected. We haven't finalized how the layer-1 prices would get relayed (trustlessly) to Starknet but we will likely implement it via storage proof. However, we decided to launch the protocol with a limit on borrowable amounts first and implement the protection middleware afterward. With the borrowing limit, we can ensure any Oracle issue wouldn't be catastrophic to the platform.

6.3 [Medium] No action is taken when oracle prices are outdated

File(s): oracles/EmpiricOracleAdapter.cairo

Description: There are two price getter functions `get_price(...)` and `get_price_with_time(...)` in the oracle contract. The market contract only uses the `get_price(...)` function for price data. Since the price `update_time` is not returned and there are no checks for the last price update it is possible for stale prices to be read by the protocol which could cause unexpected behavior. The two locations where `get_price(...)` is used are shown below.

```
1 /src/zklend/internals/Market/functions.cairo : L773, L776
```

Recommendation(s): The protocol must be defensive against malfunctions of external entities. Consider implementing a mechanism to prevent out-of-date Oracle price data from being used by the protocol.

Status: Acknowledged

Update from the client: We agree that it's important to protect against oracle malfunction/exploits, and a simple staleness check won't suffice. We plan to implement Oracle protection with a sanity range check with prices from layer-1 Uniswap pools to make sure out-of-range prices reported from the Oracle network are rejected. We haven't finalized how the layer-1 prices would get relayed (trustlessly) to Starknet but we will likely implement it via storage proof. However, we decided to launch the protocol with a limit on borrowable amounts first and implement the protection middleware afterward. With the borrowing limit, we can ensure any Oracle issue wouldn't be catastrophic to the platform.

6.4 [Low] Application ignores the number of sources used for composing the exchange price

File(s): oracles/EmpiricOracleAdapter.cairo

Description: The function `get_data(...)` is ignoring the `num_sources_aggregated` returned by `IEmpiricOracle.get_value(...)`. The number of sources is important to prevent biased data.

```
func get_data{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_checki_ptr}() -> (
    price : felt, update_time : felt
):
    alloc_locals

    let (oracle_addr) = oracle.read()
    let (pair_key) = pair.read()

    ###############################################################################
    # audit-note Not using the parameter "num_sources_aggregated". There should be a
    #            minimal number of sources in order to consider the data as valid.
    ###############################################################################
    let (price, decimals, last_updated_timestamp, _) = IEmpiricOracle.get_value(
        contract_address=oracle_addr, key=pair_key, aggregation_mode='median'
    )

    let (scaled_price) = scale_price(price, decimals)
    return (price=scaled_price, update_time=last_updated_timestamp)
end
```

Recommendation(s): The application must be defensive against malfunctions of external entities. According to the Empiric documentation, the field `num_sources_aggregated` is the number of sources aggregated in the final answer. Use this to check if one of the sources you requested was not available, or if there are enough data reports for you to rely on the answer.

Status: Acknowledged

Update from the client: We agree that it's important to protect against oracle malfunction/exploits, and a simple staleness check won't suffice. We plan to implement Oracle protection with a sanity range check with prices from layer-1 Uniswap pools to make sure out-of-range prices reported from the Oracle network are rejected. We haven't finalized how the layer-1 prices would get relayed (trustlessly) to Starknet but we will likely implement it via storage proof. However, we decided to launch the protocol with a limit on borrowable amounts first and implement the protection middleware afterward. With the borrowing limit, we can ensure any Oracle issue wouldn't be catastrophic to the platform.

6.5 [Info] Missing check for `ZToken` market address in `add_reserve(...)`

File(s): internals/Market/storage.cairo

Description: The function `add_reserve(...)` assumes that the ZToken being added has its `market` storage variable set to the address of the market contract. However, this assumption is not verified. If a ZToken with an incorrect `market` contract address is added, the market contract will be unable to mint, burn and move the ZTokens causing the market to fail. A reserve cannot be removed once added and it is not possible to have two reserves with the same underlying. This means that if a ZToken is added with an incorrect `market` address it will not be possible to deploy another reserve for the underlying asset (unless the proxy implementation is changed to remove the restrictions mentioned prior).

Recommendation(s): Consider adding a check to the `add_reserve(...)` function to ensure that the ZToken's `market` address is the market contract address (its own address). A getter would need to be added to the ZToken contract to achieve this.

Status: Acknowledged

Update from the client: We agree that an additional sanity check here would be helpful. Ideally, the `ZToken` contract should be deployed by the `Market` contract itself, but the use of the proxy pattern made it tricky (as `Market` must now be aware of the proxy layer). We've decided to leave it as is for now, as we would simply be adding all reserves at launch in one go, and any issue would be exposed before we open it up for users. We expect any additional reserve to only be added after the Cairo 1 upgrade, so we would defer implementing this to the Cairo 1 rewrite.

6.6 [Best Practices] Comment does not match function logic

File(s): internals/ZToken/functions.cairo

Description: On L137 in the function `felt_transfer_from(...)` a comment states that "Allowances are not scaled so we can just subtract directly". This comment is incorrect as the argument `amount` actually represents a scaled amount rather than a raw amount. The function and comment is shown below.

```
func felt_transfer_from{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr}(
    sender : felt, recipient : felt, amount : felt
) -> (success : felt):

    ...

    # Allowances are not scaled so we can just subtract directly
    let (existing_allowance) = allowances.read(sender, caller)
    let (new_allowance) = SafeMath.sub(existing_allowance, amount)
    allowances.write(sender, caller, new_allowance)

    ...

    return (success=TRUE)
end
```

Recommendation(s): Consider changing the comment to match the function logic.

Status: Fixed

Update from the client: A fix has been implemented in PR /41.

6.7 [Best Practices] Emit events on contract deployment and initialization

File(s): zklend/*

Description: Events should be emitted for all significant changes to state, including contract deployment and initialization.

Recommendation(s): Consider emitting events in contract constructors and initializer functions.

Status: Acknowledged

Update from the client: We agree that having these events might help with off-chain indexing. However, we believe we already have sufficient events for that, so no action would be taken at this moment.

6.8 [Best Practices] Missing argument validations in transfer functions

File(s): internals/ZToken/functions.cairo

Description: Transfer functions that make a call to the function `Internal.transfer_internal(...)` do not perform input validation on various different input arguments intended to be user addresses. The function and the corresponding arguments in which no input validation is performed are shown below:

```
felt_transfer : recipient
felt_transfer_from : sender, recipient, caller
felt_approve : spender
transfer_all : recipient
move : from_account, to_account
```

This means that zero values may be used as the argument inputs which could lead to accidental token burn or other unexpected behavior when `transfer_internal` is called.

Recommendation(s): Consider adding zero checks inside the `Internal.transfer_internal` function for the `from_address` and `to_address` arguments to ensure the neither address is zero and prevent unexpected behavior or tokens from getting burnt while performing a transfer of funds.

Status: Acknowledged

Update from the client: We agree that it would be nice to prevent the accidental transfer to the zero address (e.g. software bug), but have decided to take no action at the moment, and to defer it to the Cairo 1 re-write.

6.9 [Best Practices] Missing events emission in `set_token_source(...)`

File(s): `PriceOracle.cairo`

Description: The contract `PriceOracle.cairo` presents functions to interact with external oracles. According to inline comments, this contract is intended to be "a central oracle hub for connecting to different upstream oracles and exposing a single getter to the core protocol". There is only one external Oracle source for each token. The function `set_token_source(...)` sets the external oracle for each asset but does not emit events. Setting or changing oracles for an asset is a significant action and events should be emitted for all significant changes to the state.

Recommendation(s): Consider emitting an event upon setting or changing a token oracle source in the function `set_token_source(...)`.

Status: Fixed

Update from the client: A fix has been implemented in PR /42.

6.10 [Best Practices] Missing safety features in privileged address and implementation changes

File(s): `zklend/*`

Description: Functions that make changes to privileged addresses do not contain any safety features to prevent incorrect or malicious addresses from being set to a privileged role. The same applies to functions that make changes to an implementation class hash. In the case that an incorrect or malicious value is passed to these functions, these changes may be irreversible or affect the protocol in unexpected ways. A list of functions with missing safety features are listed below.

```
1 ZToken.upgrade, ZToken.transfer_proxy_admin, Market.transfer_ownership
```

Recommendation(s): Consider implementing safety checks to mitigate an incorrect or malicious value being set for privileged roles or implementations. These may include a) zero-checks; b) set-then-claim; c) set-then-confirm; and, d) timelock. If you wish to allow a zero address to be passed for a privileged role (for example: renouncing/ ownership), consider implementing a dedicated function for this purpose.

Status: Acknowledged

Update from the client: With the introduction of the new `replace_class` syscall for regenesis, we've decided to revamp the upgradeability pattern with this new syscall. It's confirmed by StarkWare that the `replace_class` syscall will stay after the regenesis, so it's safe to replace the proxy pattern with the use of `replace_class`. Even if this changes, we can still safely switch back to the proxy pattern before the syscall gets removed. We understand that the upgradeability revamp would be out of scope for this audit, so here we would like to simply indicate that no action will be taken.

6.11 [Best Practices] Unused functions

File(s): `internals/Market/storage.cairo`

Description: The contract `storage.cairo` contains two functions which are not interacted with at any point within the protocol: `write_reserve_factor(...)` and `write_liquidation_bonus(...)`.

Recommendation(s): Consider whether these functions need to remain within the codebase.

Status: Fixed

Update from the client: A fix has been implemented in PR /40.

6.12 [Best Practices] Variable naming for ZToken amounts can be unclear

File(s): `zklend/*`

Description: Variables and arguments that represent ZToken amounts sometimes have "raw" or "scaled" in the name to indicate whether an amount is raw or scaled. There are also points in the code where they are only named "amount", which affects readability as a reader must follow the function calls to understand what type of value an "amount" variable represents.

Recommendation(s): To improve the overall readability and code clarity, consider having "raw" or "scaled" in the name of all variables and arguments that represent ZToken amounts.

Status: Acknowledged

Update from the client: We intend to hide the "scaling" aspect of each contract as their implementation details, and external actors should simply always use scaled amounts. This is true even between our own internal contracts, with a notable exception of the `get_raw_total_supply()` function in `ZToken`, which exposes the raw supply amount. That's why we decided to simply name all our external `amount` parameters as just `amount`. For internal functions, however, we do agree that renaming those to clearly indicate whether the amounts are scaled or not could be helpful. However, since all the contracts will be rewritten in Cairo 1, we have decided to take no action on this at the moment and will implement the name change during the rewrite.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation plays a critical role in software development, as it enables effective communication between developers, testers, users, and other stakeholders involved in the software development process. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

The ZkLend Whitepaper was the primary specification used to understand the protocol. It explains all the core features (Lend, Borrow, Debt, and Flashloans) and important components (ZToken, Interest rate model, Interest accumulation, Borrowing capacity, and Liquidations). There are a total of 6 flow diagrams explaining the process for main features, which were helpful in understanding the protocol.

Variable naming led to ambiguity when dealing with ZToken amounts as some variable names would explicitly state that the amount is either "raw" or "scaled", while others would not. This sometimes made it unclear whether a variable with the name `amount` contained a "raw" or "scaled" amount value. This code clarity concern was especially apparent for function arguments where it was necessary to check the caller to determine what amount type the input is. The code could benefit from improved commenting to make functions clearer to understand. Following a NatSpec-style commenting structure for each function is recommended to make function purpose, inputs, and outputs clear to readers. During the review, it was sometimes necessary to backtrack through function calls to understand what an expected argument represents, which could be addressed with this commenting structure.

8 Test Suite Evaluation

One of the essential aspects of smart contracts is testing. Testing ensures that the contract behaves as expected, meets the required standards, and is free from errors. There are several types of tests that can be performed on smart contracts, including unit tests, integration tests, functional tests, and security tests.

- **Unit tests** involve testing individual functions or modules within the smart contract code to ensure that they work as expected. These tests can help identify bugs or errors in the code and ensure that each function performs its intended task correctly;
- **Integration tests**, on the other hand, test the interactions between different components of the smart contract. This helps ensure that the code functions correctly when integrated with other parts of the system;
- **Integration tests**, on the other hand, test the interactions between different components of the smart contract. This helps ensure that the code functions correctly when integrated with other parts of the system;
- **Functional tests** test the overall functionality of the smart contract, ensuring that it meets the intended specifications and requirements. These tests can help identify issues that may not have been caught by unit or integration tests;
- **security tests** help identify vulnerabilities in the smart contract code that could be exploited by malicious actors. This is crucial for smart contracts that manage significant amounts of money or assets.

Testing is a crucial step in the development of smart contracts. It helps ensure that the code behaves as expected, meets required standards, and is free from errors and vulnerabilities. By performing different types of tests, developers can ensure that smart contracts are secure, reliable, and functional.

The protocol presents an extensive test suite using `pytest` files directly rather than relying on a smart contract development framework. This approach to development and testing was taken to stay flexible as StarkNet and Cairo continues to evolve. To make testing easier there are two ways to conduct tests: a) Run tests scripts directly; or b) Run test suite inside a Docker container.

The test suite covers all core features of the protocol and explores edge cases well, although full end-to-end tests should be considered as this may have helped to detect issues like 5.2.1 where the tests succeed but other parts of the protocol may be affected in ways that can only be noticed after some time.

The test output shown in this section was created by running the scripts directly with the following command: `pytest -n 16 -v ./tests/*_test.py ./tests/**/*_test.py`. The test suite was also ran through Docker and the output is consistent between both testing methods.

8.1 Contracts Compilation

```
(env) dev@dev /zklend-v1-core$ ./scripts/compile.sh
Compiling zklend-v1-core/src/zklend/oracles/EmpiricOracleAdapter.cairo
Compiling zklend-v1-core/src/zklend/PriceOracle.cairo
Compiling zklend-v1-core/src/zklend/libraries/Math.cairo
Compiling zklend-v1-core/src/zklend/libraries/SafeDecimalMath.cairo
Compiling zklend-v1-core/src/zklend/libraries/SafeCast.cairo
Compiling zklend-v1-core/src/zklend/libraries/SafeMath.cairo
Compiling zklend-v1-core/src/zklend/internals/ZToken/functions.cairo
Compiling zklend-v1-core/src/zklend/internals/ZToken/events.cairo
Compiling zklend-v1-core/src/zklend/internals/ZToken/storage.cairo
Compiling zklend-v1-core/src/zklend/internals/Market/functions.cairo
Compiling zklend-v1-core/src/zklend/internals/Market/events.cairo
Compiling zklend-v1-core/src/zklend/internals/Market/structs.cairo
Compiling zklend-v1-core/src/zklend/internals/Market/storage.cairo
Compiling zklend-v1-core/src/zklend/ZToken.cairo
Compiling zklend-v1-core/src/zklend/irms/DefaultInterestRateModel.cairo
Compiling zklend-v1-core/src/zklend/Market.cairo
Compiling zklend-v1-core/src/zklend/Proxy.cairo
Compiling zklend-v1-core/src/zklend/interfaces/IPriceOracle.cairo
Compiling zklend-v1-core/src/zklend/interfaces/third_parties/IEmpiricOracle.cairo
Compiling zklend-v1-core/src/zklend/interfaces/callback/IZklendFlashCallback.cairo
Compiling zklend-v1-core/src/zklend/interfaces/IZToken.cairo
Compiling zklend-v1-core/src/zklend/interfaces/IMarket.cairo
Compiling zklend-v1-core/src/zklend/interfaces/IInterestRateModel.cairo
Compiling zklend-v1-core/src/zklend/interfaces/IPriceOracleSource.cairo
```

8.2 Tests Output

```
(env) dev@dev:~/dev/nm/synnax/gauss/NM-0058/zklend-v1-core$ pytest -n 16 -v ./tests/*_test.py ./tests/**/*_test.py
============================================= test session starts ==============================================
platform linux -- Python 3.7.12, pytest-7.1.2, pluggy-1.0.0 -- /env/bin/python3
cachedir: .pytest_cache
rootdir: /zklend-v1-core
plugins: forked-1.4.0, asyncio-0.19.0, web3-5.30.0, xdist-2.5.0, typeguard-2.13.3
asyncio: mode=strict
[gw0] linux Python 3.7.12 cwd: /zklend-v1-core
[gw1] linux Python 3.7.12 cwd: /zklend-v1-core
[gw2] linux Python 3.7.12 cwd: /zklend-v1-core
[gw3] linux Python 3.7.12 cwd: /zklend-v1-core
[gw4] linux Python 3.7.12 cwd: /zklend-v1-core
[gw5] linux Python 3.7.12 cwd: /zklend-v1-core
[gw6] linux Python 3.7.12 cwd: /zklend-v1-core
[gw7] linux Python 3.7.12 cwd: /zklend-v1-core
[gw8] linux Python 3.7.12 cwd: /zklend-v1-core
[gw9] linux Python 3.7.12 cwd: /zklend-v1-core
[gw10] linux Python 3.7.12 cwd: /zklend-v1-core
[gw11] linux Python 3.7.12 cwd: /zklend-v1-core
[gw12] linux Python 3.7.12 cwd: /zklend-v1-core
[gw13] linux Python 3.7.12 cwd: /zklend-v1-core
[gw14] linux Python 3.7.12 cwd: /zklend-v1-core
[gw15] linux Python 3.7.12 cwd: /zklend-v1-core
[gw0] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw1] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw2] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw3] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw6] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw5] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw4] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw7] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw8] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw10] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw9] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw12] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw11] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw15] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw13] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
[gw14] Python 3.7.12 (default, Jun 21 2022, 21:16:01)  -- [GCC 10.2.1 20210110]
gw0 [45] / gw1 [45] / gw2 [45] / gw3 [45] / gw4 [45] / gw5 [45] / gw6 [45] / gw7 [45] / gw8 [45] / gw9 [45] / gw10 [45] /
↪  gw11 [45] / gw12 [45] / gw13 [45] / gw14 [45] / gw15 [45]
scheduling tests via LoadScheduling

tests/Market_test.py::test_token_transferred_on_deposit
tests/Market_test.py::test_new_reserve_event
tests/DefaultInterestRateModel_test.py::test_borrow_rates
tests/Market_test.py::test_token_burnt_on_withdrawal
tests/Market_test.py::test_deposit_transfer_failed
tests/Market_test.py::test_disabling_already_disabled_collateral
tests/Market_test.py::test_borrow_token
tests/Market_test.py::test_no_debt_accumulation_without_loan
tests/Market_test.py::test_rate_changes_on_withdrawal
tests/Market_test.py::test_repay_all_with_interest
tests/Market_test.py::test_rate_changes_on_deposit
tests/Market_test.py::test_interest_accumulation
tests/Market_test.py::test_debt_repayment
tests/Market_test.py::test_cannot_withdraw_with_zero_amount
tests/Market_test.py::test_debt_accumulation
```

```
[gw1] [  2%] PASSED tests/DefaultInterestRateModel_test.py::test_borrow_rates
tests/Market_test.py::test_event_emission
[gw2] [  4%] PASSED tests/Market_test.py::test_new_reserve_event
tests/Market_test.py::test_flashloan
[gw5] [  6%] PASSED tests/Market_test.py::test_deposit_transfer_failed
tests/ZToken_test.py::test_transfer_should_emit_events
[gw3] [  8%] PASSED tests/Market_test.py::test_disabling_already_disabled_collateral
tests/ZToken_test.py::test_meta
[gw4] [ 11%] PASSED tests/Market_test.py::test_cannot_withdraw_with_zero_amount
tests/ZToken_test.py::test_transfer_all_should_emit_events
[gw13] [ 13%] PASSED tests/Market_test.py::test_no_debt_accumulation_without_loan
tests/libraries/SafeCast_test.py::test_uint256_to_felt
[gw13] [ 15%] PASSED tests/libraries/SafeCast_test.py::test_uint256_to_felt
tests/libraries/SafeMath_test.py::test_add_overflow
[gw12] [ 17%] PASSED tests/Market_test.py::test_interest_accumulation
tests/libraries/Math_test.py::test_shl
[gw10] [ 20%] PASSED tests/Market_test.py::test_rate_changes_on_deposit
tests/ZToken_test.py::test_transfer_all
[gw9] [ 22%] PASSED tests/Market_test.py::test_rate_changes_on_withdrawal
tests/ZToken_test.py::test_burn_all
[gw11] [ 26%] PASSED tests/Market_test.py::test_debt_accumulation
tests/libraries/Math_test.py::test_shr
[gw13] [ 28%] PASSED tests/libraries/SafeMath_test.py::test_add_overflow
tests/libraries/SafeMath_test.py::test_sub
[gw6] [ 31%] PASSED tests/Market_test.py::test_token_transferred_on_deposit
tests/ZToken_test.py::test_balance_should_scale_with_accumulator
[gw12] [ 33%] PASSED tests/libraries/Math_test.py::test_shl
tests/libraries/SafeMath_test.py::test_sub_underflow
[gw7] [ 35%] PASSED tests/Market_test.py::test_token_burnt_on_withdrawal
tests/ZToken_test.py::test_approve_should_change_allowance
[gw13] [ 37%] PASSED tests/libraries/SafeMath_test.py::test_sub
tests/libraries/SafeMath_test.py::test_div_division_by_zero
[gw11] [ 40%] PASSED tests/libraries/Math_test.py::test_shr
tests/libraries/SafeMath_test.py::test_div
[gw14] [ 42%] PASSED tests/Market_test.py::test_debt_repayment
tests/libraries/SafeCast_test.py::test_uint256_to_felt_invalid_uint256
[gw12] [ 44%] PASSED tests/libraries/SafeMath_test.py::test_sub_underflow
[gw14] [ 46%] PASSED tests/libraries/SafeCast_test.py::test_uint256_to_felt_invalid_uint256
[gw15] [ 48%] PASSED tests/Market_test.py::test_repay_all_with_interest
tests/libraries/SafeCast_test.py::test_felt_to_uint256
[gw13] [ 51%] PASSED tests/libraries/SafeMath_test.py::test_div_division_by_zero
[gw11] [ 53%] PASSED tests/libraries/SafeMath_test.py::test_div
[gw15] [ 55%] PASSED tests/libraries/SafeCast_test.py::test_felt_to_uint256
[gw1] [ 57%] PASSED tests/Market_test.py::test_event_emission
tests/libraries/SafeCast_test.py::test_uint256_to_felt_out_of_range
[gw1] [ 60%] PASSED tests/libraries/SafeCast_test.py::test_uint256_to_felt_out_of_range
[gw8] [ 62%] PASSED tests/Market_test.py::test_borrow_token
tests/ZToken_test.py::test_transfer_from
[gw5] [ 64%] PASSED tests/ZToken_test.py::test_transfer_should_emit_events
tests/libraries/SafeDecimalMath_test.py::test_mul_overflow
[gw3] [ 66%] PASSED tests/ZToken_test.py::test_meta
tests/libraries/SafeDecimalMath_test.py::test_div
[gw4] [ 68%] PASSED tests/ZToken_test.py::test_transfer_all_should_emit_events
tests/libraries/SafeMath_test.py::test_add
[gw5] [ 71%] PASSED tests/libraries/SafeDecimalMath_test.py::test_mul_overflow
[gw3] [ 73%] PASSED tests/libraries/SafeDecimalMath_test.py::test_div
[gw4] [ 75%] PASSED tests/libraries/SafeMath_test.py::test_add
[gw10] [ 77%] PASSED tests/ZToken_test.py::test_transfer_all
tests/libraries/SafeMath_test.py::test_mul
[gw9] [ 80%] PASSED tests/ZToken_test.py::test_burn_all
tests/libraries/SafeMath_test.py::test_mul_felt_overflow
[gw6] [ 82%] PASSED tests/ZToken_test.py::test_balance_should_scale_with_accumulator
[gw10] [ 84%] PASSED tests/libraries/SafeMath_test.py::test_mul
[gw7] [ 86%] PASSED tests/ZToken_test.py::test_approve_should_change_allowance
[gw9] [ 88%] PASSED tests/libraries/SafeMath_test.py::test_mul_felt_overflow
[gw8] [ 91%] PASSED tests/ZToken_test.py::test_transfer_from
[gw2] [ 93%] PASSED tests/Market_test.py::test_flashloan
tests/libraries/SafeDecimalMath_test.py::test_mul
[gw2] [ 95%] PASSED tests/libraries/SafeDecimalMath_test.py::test_mul
[gw0] [ 97%] PASSED tests/Market_test.py::test_liquidation
tests/libraries/SafeMath_test.py::test_mul_uint256_overflow
[gw0] [100%] PASSED tests/libraries/SafeMath_test.py::test_mul_uint256_overflow
```

```
============================================== warnings summary ===============================================
../env/lib/python3.7/site-packages/frozendict/__init__.py:16: 16 warnings
  /env/lib/python3.7/site-packages/frozendict/__init__.py:16: DeprecationWarning: Using or importing the ABCs from
   ↪  'collections' instead of from 'collections.abc' is deprecated since Python 3.3,and in 3.9 it will stop working
    class frozendict(collections.Mapping):

tests/Market_test.py::test_flashloan
  /zklend-v1-core/tests/Market_test.py:1595: RuntimeWarning: coroutine 'Account.execute' was never awaited
    105 * 10**18,  # return_amount
  Enable tracemalloc to get traceback where the object was allocated.
  See https://docs.pytest.org/en/stable/how-to/capture-warnings.html#resource-warnings for more info.

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
================================== 45 passed, 17 warnings in 1162.93s (0:19:22) ===================================
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology 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 the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.