zkLend Formal Specification Report

1 Introduction

This report presents a formalisation in the Coq proof assistant[2] of the zkLend money-market protocol, as described in the whitepaper [3], informed by the implementation at the following commit hash:

hash	fe7f522966a1df3ca0c59190a1a74b2d1c8b12af

In section 2, we first introduce an abstract model of the zkLend money-market protocol's state. In section 3 we introduce the invariants of the protocol. These invariants underlay the protocol's correctness and enforce the desired security properties. Once we have specified the operations of the zkLend protocol's Market and ZToken contracts and introduced their Coq formalisations in section 4, we introduce a Coq proof of correctness of all the operations with respect to the previously introduced invariants in 5. Finally, in section 6, we introduce some limitations of the model regarding our reference implementation.

2 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.

2.1 State representation

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

- `Token`: The set of tokens accepted by the protocol.
- `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, `State`, consists of a record with members:

- $erc20\_balances$: `Token` $\rightarrow$ `User` $\rightarrow \mathbb{Q}^{\geq 0}$: defines a mapping to store the token balance for a given user or the market itself.

- $erc20\_allowances$: `Token` $\rightarrow$ `User` $\rightarrow$ `User` $\rightarrow \mathbb{Q}^{\geq 0}$: defines a mapping which stores the allowances from a token from a user or market to another user or the market itself.
- $z\_balances$: `Token` $\rightarrow$ `User` $\rightarrow$ `Time` $\rightarrow \mathbb{Q}^{\geq 0}$: defines a mapping which stores the z-balances value for a token and user in a given point of time.
- $z\_allowances$: `Token` $\rightarrow$ `User` $\rightarrow$ `User` $\rightarrow \mathbb{Q}^{\geq 0}$: stores the z-allowance value for a given user and token.
- $debts$: `Token` $\rightarrow$ `User` $\rightarrow$ `Time` $\rightarrow \mathbb{Q}^{\geq 0}$: defines a mapping which stores the debt of a user given a token at a given point of time.
- $collateral$: `User` $\rightarrow \mathcal{P}$(`Token`): stores the set of tokens that can be used as collateral for a given user.
- $owner$: `User` $\cup \{none\}$: 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` $\rightarrow [0, 1]$: current borrow factor for a given token.
- $collateral\_factor$: `Token` $\rightarrow [0, 1]$: current collateral factor for a given token.
- $flash\_loan\_fee$: `Token` $\rightarrow \mathbb{Q}^{\geq 0}$: current fee for flash loans for a given token.
- $liquidation\_bonus$: `Token` $\rightarrow \mathbb{Q}^{\geq 0}$: current liquidation bonus for a given token.
- $reserve\_factor$: `Token` $\rightarrow [0, 1]$: current reserve factor for a given token.

The Coq implementation for state type is available online at the following link.

2.2 State Invariants

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

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

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

2.3 Model parameters

Our model uses some parameters which are not possible to express 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:

- $usd\_value \in$ `Token` $\rightarrow$ `Time` $\rightarrow \mathbb{Q}^{\geq 0}$: The price in USD of each token at a given point in time.
- $borrowing\_rate \in$ `State` $\rightarrow$ `Token` $\rightarrow$ `Time` $\rightarrow \mathbb{Q}^{\geq 0}$: interest rate on borrowings for a given token.

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

2.4 Helper functions

To specify the relevant protocol properties, we need to define some auxiliary functions. 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 of time.

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

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

The function *collateral_value* definition can be found here, and the definition for *utilisation_rate* is available here.

3 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 3.1 describes the well-formedness predicate over the protocol state, and Section 3.2 defines the solvency condition predicate.

3.1 State well-formedness

We define a well-formed state as the one which has all of 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).
```

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. 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.

3.2 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:

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

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 Operation Specifications

In this section, we present 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.

4.1 Specification of Market's Functions

4.1.1 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.

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

$$
\begin{array}{c}
state \left\{ \begin{array}{c} z\_balances(token) = old\_z\_balances, \\ erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance, \\ erc20\_allowances(token, \#caller, market) = old\_erc20\_allowance \end{array} \right\} \\
\xrightarrow{\texttt{deposit}(token, amount), \#caller, \#block\_timestamp} \\
state \left\{ \begin{array}{c} z\_balances(token) = new\_z\_balances, \\ erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance, \\ erc20\_allowances(token, \#caller, market) = new\_erc20\_allowance \end{array} \right\}
\end{array}
$$

Figure 1: Specification for the `deposit` function.

4.1.2 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.

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

$$
\begin{array}{c}
state \left\{ \begin{array}{c} z\_balances(token) = old\_z\_balances, \\ erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance \end{array} \right\} \\
\xrightarrow{\texttt{withdraw}(token, amount), \#caller, \#block\_timestamp} \\
s@state \left\{ \begin{array}{c} z\_balances(token) = new\_z\_balances, \\ erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance \end{array} \right\}
\end{array}
$$

Figure 2: Specification for the `withdraw` function.

4.1.3 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 withdraw and resets the caller's z-balance to zero in the resulting state.

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

$$
state \left\{ \begin{array}{c} z\_balances(token) = old\_z\_balances, \\ erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance \end{array} \right\}
$$

$$
\xrightarrow{\texttt{withdraw\_all}(token), \#caller, \#block\_timestamp}
$$

$$
s@state \left\{ \begin{array}{c} z\_balances(token) = new\_z\_balances, \\ erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance \end{array} \right\}
$$

Figure 3: Specification for the `withdraw_all` functions.

4.1.4 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.

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

$$
state \left\{ \begin{array}{c} erc20\_balances(token, \#caller) = old\_user\_erc20\_balance, \\ erc20\_balances(token, market) = old\_reserve\_erc20\_balance, \\ debts = old\_debts \end{array} \right\}
$$

$$
\xrightarrow{\texttt{borrow}(token, amount), \#caller, \#block\_timestamp}
$$

$$
s@state \left\{ \begin{array}{c} erc20\_balances(token, \#caller) = new\_user\_erc20\_balance, \\ erc20\_balances(token, market) = new\_reserve\_erc20\_balance, \\ debts = new\_debts \end{array} \right\}
$$

Figure 4: Specification of the `borrow` function

4.1.5 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.

4.1.6 Specification of `repay_all` function (Coq code)

The `repay_all` function allows the caller to use a specified token value to pay off all of 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

4.1.7 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 are permitted to 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 than 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.

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

$$
collateral\_amount = debt\_amount \times \frac{usd\_price(debt\_token, \#block\_timestamp)}{usd\_price(collateral\_token, \#block\_timestamp)} \times (1 + liquidation\_bonus(collateral\_token))
$$

$$
\begin{array}{l}
old\_debts(debt\_token, user, \#block\_timestamp) \geq debt\_amount \\
old\_caller\_erc20\_balance \geq debt\_amount \\
old\_z\_balances(user, \#block\_timestamp) \geq collateral\_amount
\end{array}
$$

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

$$
\sum_{t \in \texttt{Token}} (new\_debts(t, user, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)) \geq collateral\_value(s, user, \#block\_timestamp)
$$

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

Figure 7: Specification for the `liquidate` function.

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

The function `enable_collateral` marks that a specific token as available to 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`

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

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

$\#caller \in \texttt{User}$
$token \in old\_collateral$
$\sum_{t \in \texttt{Token}} \left(\frac{debts(t, \#caller, \#block\_timestamp) \times usd\_value(t, \#block\_timestamp)}{borrow\_factor(t)} \right) \leq collateral\_value(s, \#caller, \#block\_timestamp)$

$$state \{ \quad collateral(\#caller) = old\_collateral \quad \}$$
$$\xrightarrow{\texttt{disable\_collateral}(token), \#caller}$$
$$s@state \{ \quad collateral(\#caller) = old\_collateral \setminus \{token\} \quad \}$$

Figure 9: Specifications for the `disable_collateral` functions.

4.1.10 Specification of the permissioned entry points (Coq code)

All functions listed here are only callable by the owner. The `add_reserve` functions adds 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*.

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

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

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

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

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

4.2 Specification of ZToken Functions

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

Transfer an amount of token 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.

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

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

Figure 11: Specifications for the `transfer` function.

4.2.2 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.

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

Transfer an amount 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.

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

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

Figure 13: Specifications for the `transferFrom` functions.

4.2.4 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.

5 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 prove that such functions satisfy the invariants of the first solvency condition. We present below, as an example, only two of such solvency condition proof: *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 1: 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 2: Solvency condition 1 proofs for ZToken contract operations

6 Limitations

Note that this model has some limitations with respect to modelling an implementation of the zkLend money-market protocol:

- 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.
- 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.

7 Conclusions

This report has introduced a formalised 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, as well as 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 formalised them. Finally, we have introduced formalised proofs that our abstract specification satisfies the well-formedness invariant as well as 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].

8 Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the formal specification outlined in 1. Introduction. The results set out in this report may have limitations and may not be complete. Nethermind has provided the formal specification 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 specification is limited to the protocol defined in 1. Introduction, with limitations as described in 6. Limitations. 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.

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.

[2] *The coq proof assistant*. URL: `https://coq.inria.fr/`.

[3] *zkLend whitepaper*. URL: `https://zklend.com/documents/2022.12.02_zkLend-White-Paper.pdf`.