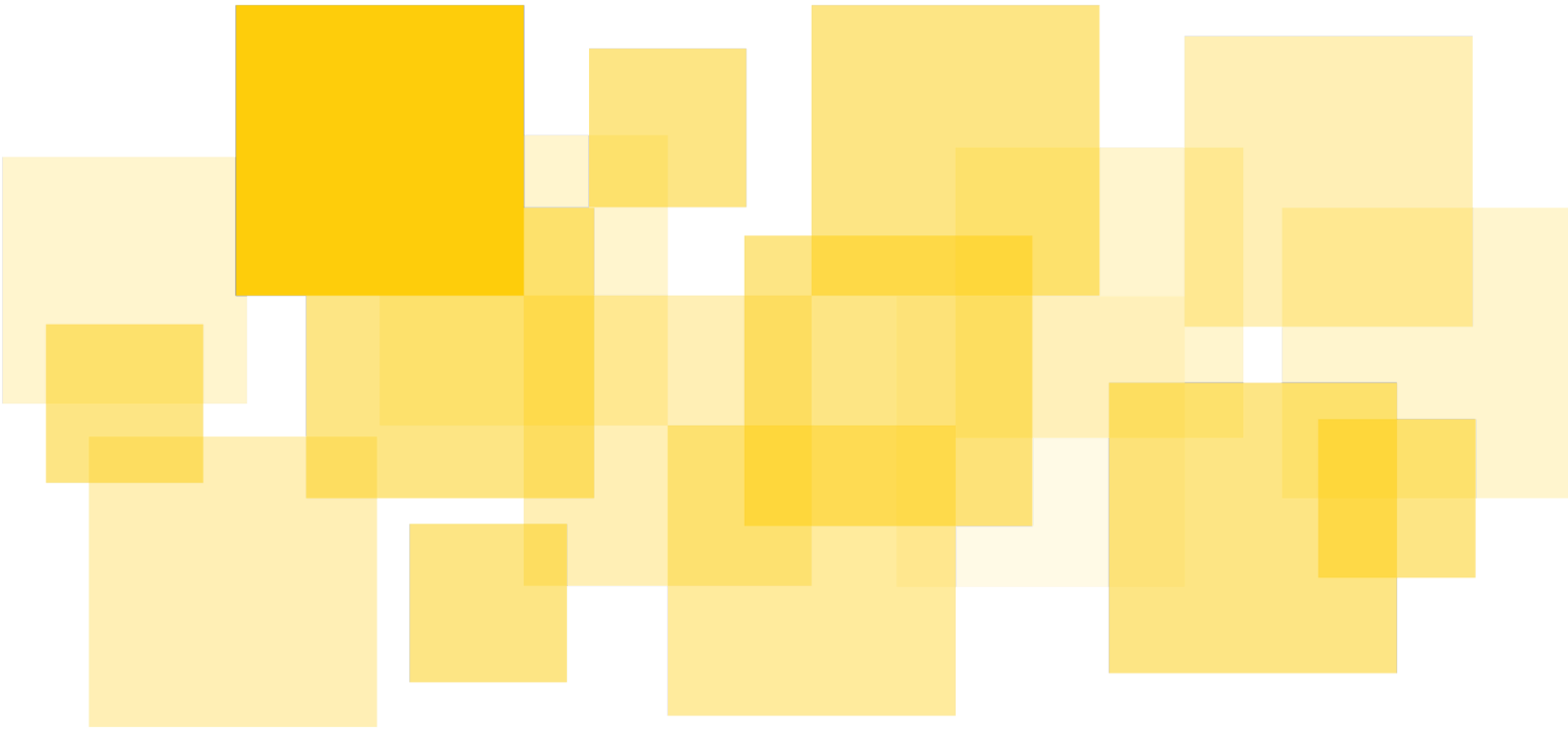
Protocol Design Review Summary

Folks.Finance

Delivered: September 27th, 2021



Prepared for Folks.Finance by

runtime verification

Table of Contents

Overview

Folks.Finance engaged Runtime Verification Inc to conduct an initial two-week design review of the Folks.Finance protocol, which specifies a decentralized, algorithmic lending application to be built on the Algorand blockchain. This report summarizes what the review involved and some of the important discussions and comments made in the process.

Objective

The main objective of this review was to develop a deeper understanding of the business logic of the application and prepare for an upcoming full-blown security audit of the implementation of the protocol as smart contracts written in TEAL. The review presented an opportunity to discuss various aspects of the design and to make some suggestions for improvement, which we summarize here in this report.

Scope

The review was conducted by Musab Alturki over the period spanning Sep 6, 2021 through Sep 22, 2021.

The scope of the review was primarily limited to the Folks.Finance design document (white paper) titled “Folks.Finance Project”, which gives a high-level description of the underlying protocol’s design and anticipated interactions with the application. We also had access to an early version of the implementation of the protocol as smart contracts in TEAL.

Methodology

The review involved studying the provided artifacts carefully, and discussing various aspects of the protocol with the Folks.Finance team and making suggestions for improvement as appropriate. Overall, the protocol was found to be very well thought out and the design document to be well structured and written.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contracts only, and make no material claims or guarantees concerning the contracts' operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of these contracts.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Findings

This section summarizes the important discussion points raised during the review. None of these points represented a design flaw or a security vulnerability in the system. Rather, they were suggestions to improve the protocol's efficiency and robustness.

1. Observations Regarding How Fast the Interest Indices Grow

The borrow and deposit interest indices monotonically increase with time. There was initially a question of how fast these indices grow, and if the growth rate could potentially be too high that would eventually cause overflow in the long run (especially the borrow interest index since it's defined using an exponential equation). The Folks.Finance team had already investigated this concern before and performed various simulations of the protocol, which showed that, under normal conditions including those involving high volumes of lending operations, the growth rates remain within reasonable values that won't pose arithmetic problems for any extended time period of operations (in tens or even hundreds of years).

The formula for the borrow interest index (the corrected version of eq. 11, page 13), which is shown below:

$$I_{b_t} = (1 + \alpha_{b_{t-1}})^{\Delta t} * I_{b_{t-1}}$$

was meant to over-approximate the actual index by compounding the unpaid interest exponentially (instead of linearly) with time to guard against having the index fall below the deposit interest index due to truncation in integer division.

However, a closer look at the exponential equation revealed a potential issue that could arise in the other extreme case when no frequent interactions with a pool are taking place, so that Δt is larger than usual. In practice, Δt will typically be equal to the block production time, assuming there is at least one interaction with the pool at every block (which is about 4.5 seconds in Algorand), which seems fine. But in case of a less-than-normal activity with the pool or in case of network congestion, protocol state updates may begin to skip blocks, and so Δt increases (in multiples of block time). This means that the borrow index will begin to grow much larger than the deposit index as long as these operation conditions persist, potentially causing fixed-point arithmetic issues in practice.

The initial recommendation was to add a (constant) term ϵ to compensate for any potential arithmetic rounding errors in an alternative linear-growth equation:

$$I_{b_t} = (1 + \epsilon + \alpha_{b_{t-1}} * \Delta t) * I_{b_{t-1}}$$

where ϵ is ideally a very small positive value (much smaller than 1). Therefore, being a linear equation avoids the exponential growth that the original formula may exhibit when Δt is larger than a few seconds, while at the same time over-approximates the actual value protecting against under-calculated borrow interests.

Still, the Folks.Finance team pointed out another potential issue with the suggestion above, which is that the formula always increases the value of the index even if there are no updates to the pool (i.e. Δt is zero). The suggestion was to use instead an $\epsilon > 1$ as a multiplicative factor:

$$I_{b_t} = (1 + \epsilon * \alpha_{b_{t-1}} * \Delta t) * I_{b_{t-1}}$$

So ϵ here is a multiplier that would ideally have a value slightly greater than 1 (to ensure any rounding errors are compensated for). This ensure that the index:

1. Grows only linearly with time even during extremely low activity (Δt is relatively large)
2. Does not grow in the absence of any activity (Δt is zero)
3. Always over-approximates the actual value of the index, but only by a small margin (since ϵ is only slightly greater than 1)

Finally, we recommend conducting further simulations to validate these claims above using different values for the interest α_t and $0 \leq \Delta t \leq kT$, where T is the block time and k is a positive whole factor representing the number of skipped blocks.

2. Implementing State Updates

It is perfectly appropriate for the high-level description of the protocol given in the white paper to update the state at every protocol state change (psc), which is defined (in most cases) as a state change that is caused by processing a deposit, borrow, repay or

liquidate operation. These operations change the utilization rate and hence the borrow and deposit interests.

Implementing this strategy in practice, however, can be both dangerous and wasteful since transactions are grouped in blocks and these transactions are validated at the same block time. So when multiple protocol interactions (e.g. a borrow and a deposit) occur in the same block, the protocol state is unnecessarily updated multiple times in the same block. Furthermore, intra-block state updates make the protocol more vulnerable to interest manipulation attacks, in ways similar to how automated market makers (AMMs) can be vulnerable to price manipulation attacks (e.g. the original Uniswap V1 was vulnerable to price manipulation due to same-block state updates of prices, which was fixed in Uniswap V2 -- see the Uniswap V2 white paper).

This can be avoided by implementing state updates so that they are performed at most once per block, which can be achieved by making updates conditional on whether the state has already been updated in the current block. If so, the protocol would just use the current state without updating it. So in other words, only the first transaction in a block that represents a psc would update the state. All other transactions in the same block would simply use the updated state.

3. Using Pooled Transaction Fees to Simplify Fee Management

It used to be the case before that Algorand required the account who signs a transaction to pay the transaction fee, which meant that in a transaction group where a contract account is supposed to sign messages, some other account(s) would need to repay the contract account back to refund the transaction fees. This is no longer the case now with the introduction of pooled transaction fees (see "Pooled Transaction Fees" in this article).

So essentially, a transaction group can have one transaction in it covering the fees of all or some of the fees of other transactions in the group. The only requirement is that the sum of all fees paid by all transactions in the group are at least as large as the total fees that need to be paid by the group, regardless of which transaction is actually paying.

Therefore, in the design of transaction groups for the protocol, we may have the user (e.g. depositor in a deposit operation) pay a total sum of fees that covers his/her

transaction plus transactions signed by the contract account. The logic signature of the contract account can check that both:

1. Its current logicsig-signed transaction is paying 0 fees, and
2. The user-signed transaction is paying fees high enough to cover all transactions in the transaction group.

4. Increasing Fault-tolerance Using Multiple Different Oracles

When performing a lending operation, a price oracle needs to be consulted for the current exchange rate of a particular pair of assets. In the case that the exchange rate hasn't been updated recently enough, the lending contract will refuse using the rate, and in this case the transaction gets rejected.

Since the operation of the protocol (particularly borrows and liquidations) depends on reading these values frequently, this could potentially be a viable DoS attack vector, as the price oracle represents a single point of failure. This is a known problem, typically referred to as “the Oracle problem”. The oracle may also be manipulated resulting in rates that are excessively high or low. Without the price oracle or when the price oracle gets compromised, the operation of the protocol could be severely hindered.

One way to mitigate this risk is to use multiple different price oracles (switching across them in cases of failure). An additional mitigation measure is to set a rate tolerance level/percentage beyond which the published value is rejected.

5. Liquidity-providing program

Liquidity is essential for the proper operation of the protocol. The liquidity-providing program aims to incentivize providers with FOLKS rewards. However, in this program, the provider cannot borrow against their deposits because they are not issued “ff_tokens” that can be locked for collateral. This is by design. The Folks.Finance team was studying adjusting the design in a way that would allow liquidity providers to borrow against their deposits, as this could potentially be very beneficial for the protocol.

The recommendation here is to keep the current design as is, where the liquidity-providing program is kept separate from lending pools. Two fundamental

reasons why we believe this is the best approach (at least at this early stage of development) are as follows:

1. Current best practices in designing lending platforms keep these functions separate (as far as we are aware). A liquidity provider (LP) commits a certain amount to the protocol for a pre-specified period of time. If the LP decides to borrow, he/she can always participate in lending pools and borrow against a deposited collateral. An LP may plan ahead by allocating a portion of the funds to the liquidity program while the other portion is used to fund lending pools.
2. This separation of concerns means simpler logic to implement in smart contracts, which is not only helpful for ease and convenience of implementation but also (more importantly) means easier logic to analyze and reason about formally, and thus more secure.

6. Implementing Flash Loans

Flash loans provide an interesting functionality to lending platforms, but are intrinsically tricky to reason about and implement properly. This is mainly due to the fact that the semantics of a flash loan depends on that of interactions with external parties/systems. Furthermore, as the white paper explains, properly implementing flash loans requires proper support from the underlying blockchain platform (e.g. using transaction sub-groups) and cooperation with external third parties.

Our recommendation here is to not support flash loans at this early stage of development and postpone it until a future release of the product due to the following reasons:

1. Flash loans require support from the underlying Algorand blockchain (perhaps in the form of transaction subgroups or contract call opcodes in TEAL) which is currently not present. Moreover, when this support is first rolled out, it may potentially increase the attack space of the protocol enabling interactions that were not possible before. A careful investigation of what this support entails will be needed before it's used.
2. Flash loans require the presence of reliable oracles, more so than regular loans, which may not be available abundantly enough at the moment.
3. Flash loans require coordination with other service providers (other lending platforms, AMMs, staking services, … etc) to be effective or attractive to users.

This coordination can be worked on during the current development stage while the ecosystem evolves and matures.

7. Whitelisting of Markets

The creation of lending pools can either be permissioned, so that only an admin is authorized to create pools, or permissionless, so that any user can create a pool. For a lending platform such as Folks.Finance, the creation of pools should be permissioned so that the lending protocol remains predictable and secure. We recommend adding a few statements about this in the white paper as it would help set users' expectations and increase users' confidence in the system's trustworthiness.

Furthermore, when whitelisting pools, it will be important to consider the type of the asset for which the pool is created and how the asset is defined. An asset that has its "manager", "freeze" or "clawback" addresses set, should only be supported if these addresses are trusted, since they will have control over user holdings of the asset (including the holdings of the asset in the contract account representing the lending pool). A malicious "manager", "freeze" or "clawback" address could freeze assets (so users won't be able to recover collateral) or revoke holdings of the asset to any other address. Trusted assets such as USDC, that are managed by trusted entities, should be fine, but others should be carefully scrutinized as long as they have these attributes set. An asset where the "manager", "freeze" or "clawback" addresses are not set are trustless, and thus should be fine to whitelist as well.