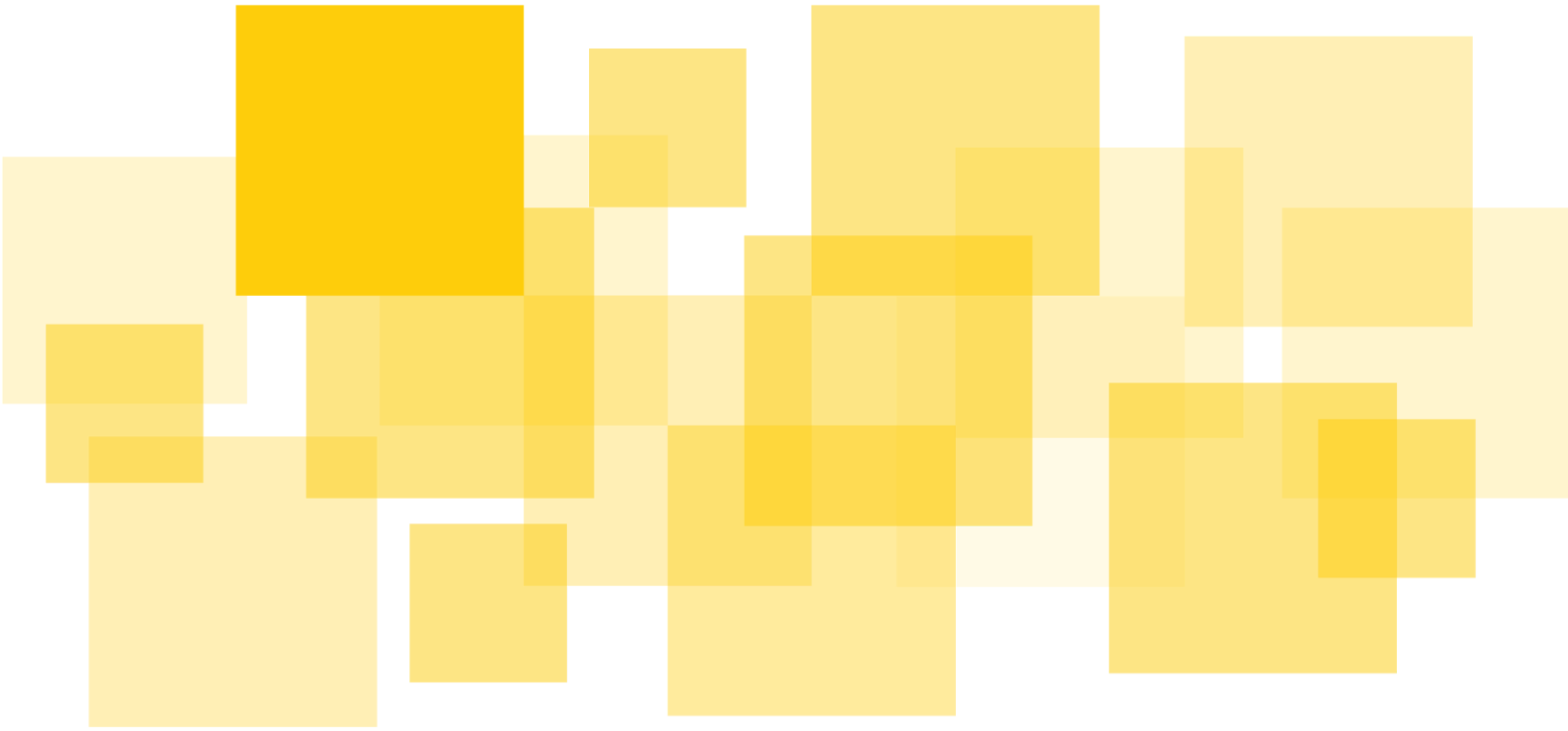
Formal Verification Report

Tezos Dexter V2 Contract

Delivered: June 9th, 2021



Prepared for the Tezos Foundation by

runtime verification

Summary

Verification Artifacts

Disclaimer

Functional Correctness of Dexter Entrypoints

Safety Property Verification

Invariants

Assumptions

Assumptions for Tezos Smart Contract Execution

Assumptions for External Contracts

Abstract Functional Specification of Dexter Entrypoints

Recommendations

A01: Input validation for `%removeLiquidity`

A02: Uniqueness of liquidity token contracts

A03: Input validation for `%setLqtAddress`

A04: Initialization of state variables

A05: Front-running possibility during pool initialization

A06: Verification of token and liquidity token contract implementations

Summary

Runtime Verification, Inc. have formally verified the Tezos Dexter V2 smart contract. The formal verification was conducted by Stephen Skeirik, Rikard Hjort, Nishant Rodrigues, and Daejun Park, from April 15, 2021 to June 4, 2021.

We formalized and proved safety properties of the Dexter contract. We also identified several requirements on external contracts for the functional correctness and security of Dexter. It is important that the users ensure that such requirements are satisfied throughout the operation of Dexter. To be more specific, we provide several example scenarios where the requirements could be violated. (See the Assumptions section for more details.)

We mechanized our proofs in the K framework to utilize the existing K Michelson semantics. We note that, however, part of our proofs have *not* been fully mechanized, which we leave as future work. The non-mechanized proofs have gone through only manual proof check, which becomes part of the trust base.

Scope

The target of the formal verification is the following smart contract source and bytecode files at git-commit-id 8a5792a5:

- dexter.mligo: Dexter contract source file written in LIGO
- dexter.mligo.tz: Compiled Michelson bytecode for FA1.2
- dexter.fa2.mligo.tz: Compiled Michelson bytecode for FA2

Note that the lqt_fa12.mligo file, the reference implementation of FA1.2 with the mint and burn features, is *not* in the scope of this formal verification engagement. Note that, however, our formal verification result of the Dexter contract is applicable for *arbitrary* token and liquidity token contract implementations, provided that they faithfully implement the requirements we explicitly formalized in the Assumptions section.

The formal verification is limited in scope within the boundary of the Dexter smart contract only. Off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are *not* in the scope of this engagement.

Approach

We adopted a refinement-based verification approach to reduce verification effort as follows.

First, we formalized “baseline” properties for each Dexter entrypoint function, that specify the storage update and the sequence of operations emitted. These baseline properties were written based on our understanding of the LIGO source code, but they were verified against the

compiled Michelson bytecode using the K Michelson semantics. This way, we could ensure that the Dexter source code was faithfully compiled to the Michelson bytecode, and the compiler does *not* introduce into the bytecode any new behaviors that were not originally intended in the source code.

Then, we formulated safety properties of Dexter over any sequence of arbitrary operations including non-Dexter operations. One of the most critical properties is that "the liquidity share price *never* decreases," where the share price refers to (the geometric mean of) the XTZ and token reserves per share. This share price preservation property implies that there is no way to illegally drain funds from the liquidity pool, and thus users' funds are safe (provided that certain conditions are met, as described in the Assumptions section).

Lastly, we verified the safety properties. To reduce verification effort for the safety properties, we first translated the baseline properties over an abstract configuration to abstract away certain details that are irrelevant to the safety properties. This enabled us to prove the safety properties using the abstract baseline properties.

About our assumptions

Since the external contracts that Dexter interacts with could be arbitrary, we had to make certain assumptions on unknown external contracts. We did our best to minimize such assumptions to make our verification result as general as possible. The assumptions were formalized as axioms and used in our proofs. We note that our verification result is valid only when the assumptions are met throughout the actual operation of Dexter.

Trust base

The verification of the baseline properties have been fully mechanized in the K framework on top of the K Michelson semantics. The trust base includes the K framework as well as the K Michelson semantics. However, the LIGO-to-Michelson compiler is *not* in our trust base, since the verification was conducted against the compiled bytecode.

The proofs for the safety properties have *not* been fully mechanized. Specifically, some proofs are *not* yet machine-checkable, but only manual proof checks have been performed. However, all the proofs except one are written in a machine-checkable form, and we believe that they can be made machine-checkable with reasonable effort, which we leave as future work.

Our safety property proofs depend on a mathematical model of the general Tezos smart contract execution process. This model includes several explicit assumptions about how Tezos smart contracts execute. Any violation of these assumptions in the actual Tezos blockchain implementation (due to implementation bugs, updates to the protocol, or an inconsistency between our mathematical model and the actual implementation) could invalidate the proofs.

Verification Artifacts

Our proof artifacts contain the full details of our formal verification process. Though we have made significant efforts to keep our proof scripts readable, given the amount of details which we must encode to complete our proofs, reading the raw proof scripts may still require concentrated effort.

Reproducibility

Interested users may wish to run our mechanized proofs on their own machines to better understand our tooling and approach. Since our mechanized proofs were performed using the K Framework and the K-Michelson semantics, both of these tools must be installed first. To install these tools, use your favorite Git client to check out our K-Michelson Git repository and then consult our installation guide. Then, to reproduce our mechanical proofs, run the following command:

```
$ make dexter-prove
```

Structure

All of the Dexter proof artifacts are included in our K-Michelson Git repository under the tests/proofs/dexter directory. We have four source files, three of which contain our mechanized proof of the baseline properties and one of which contains our high-level manual proof of some important safety properties.

- Baseline Property Proof Artifacts
    - dexter-compiled.md: contains the Michelson source code of the compiled Dexter contracts in both the FA1.2 and FA2 variants.
    - dexter.md: contains helper functions and semantic descriptions of each Dexter entrypoint.
    - dexter-spec.md: contains proof scripts for the baseline property proofs.
- High-Level Safety Properties
    - dexter-properties.md: contains the formulation and proof of high-level safety properties over arbitrary operation sequences.

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 contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

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.

Functional Correctness of Dexter Entrypoints

Summary

In this section, we survey our proofs of functional correctness of Dexter entrypoints. Our proof scripts consist of two pieces:

1. The Michelson code corresponding to the Dexter contract
2. High-level functional descriptions of each Dexter contract entrypoint derived from the Dexter LIGO code

Each script relates the high-level entrypoint description to the low-level Michelson code. Because we perform our proofs directly over Michelson code, we avoid the need to assume the correctness of the LIGO compiler.

Each functional description of a Dexter contract entrypoint specifies: (a) the conditions under which they succeed as well as how the storage gets updated; and (b) the conditions under which the transactions fail which leads the Michelson VM to revert. These entrypoint descriptions as well as the proofs relating them to the Michelson code are contained in the artifact `dexter-spec.md`. Each description covers only the execution of the Dexter contract directly including the operations that it emits; they *do not* include the execution of any internal operations emitted by the Dexter contract.

Macros

The artifact `dexter-compiled.md` contains the compiled dexter code, including the FA1.2 and FA2 versions as K macros.

Lemmas and Helper Functions

The artifact `dexter.md` contains data structures and functions that make the mechanized proof claims more readable. This includes: (a) a model of the storage of the Dexter contract; (b) constructors representing each Dexter LIGO contract entrypoint parameters; (c) a function `runProof()` which consumes a boolean indicating the Dexter FA1.2 or FA2 version and an abstract Dexter entrypoint and sets up the Michelson VM to execute that Dexter entrypoint.

It also contains a set of small (and hopefully trivially obvious) assumptions, such as $X / 1 = X$.

Safety Property Verification

The ultimate safety property we proved is the safety of the liquidity pool reserves, that is, that users' funds in the liquidity pool are safe. More specifically, we formulated the safety property "***the liquidity share price never decreases***," where the share price is defined as the geometric mean of the XTZ and token reserves per share, as follows:

the share price := \sqrt(the XTZ reserve * the token reserve) / the total shares

The share price preservation property subsumes safety properties such as:[1]

- Each entrypoint is functionally correct, never sending more assets than it should (e.g., due to rounding errors).
- It is not possible to earn "free" money by repeating token exchanges back and forth, or adding and removing liquidity back and forth.
- No re-entrancy vulnerabilities exist.
- No exploits for the non-atomicity of `%updateTokenPool` exist.

We proved that the share price preservation property holds over any sequences of arbitrary operations. The proof is based on the other Dexter invariants that we proved, assumptions on external contracts and Tezos smart contract execution, and functional specifications of the Dexter entrypoint functions, which we explain below.

The full details can be found in [dexter-properties.md].

Invariants

One of the critical invariants of Dexter is about the Dexter state variables. In Dexter, the share price as well as the token exchange rate are determined by the three state variables, `XtzPool`, `TokenPool`, and `LqtTotal`, which keep track of the XTZ reserve size, the token reserve size, and the total number of shares, respectively. However, while the Dexter entrypoint functions immediately update these state variables, the actual amount of reserves or shares will be updated later by the continuation operations emitted by the entrypoints. Since another Dexter entrypoint can potentially be reentered during the execution of the current continuation operations, and this reentrance can be repeated, the gap between the state variables and the actual reserves/shares is unbounded.

[1] Note that, however, this property has nothing to do with the market price of shares. Indeed, for example, the USD value of the share price could decrease due to the so-called "impermanent loss" problem.

To deal with the potentially unbounded gap, we formulated the gap as a function of the continuation operations, and used it to specify the invariant on the state variables. See the claim `[inv]` in dexter-properties.md for the details.

This invariant was proved by the induction on sequences of operations and case analysis over different types of operations. Most of the proof is straightforward, except the proof of `[lemma-update-token-pool-internal]` which states that whenever the `%updateTokenPoolInternal` entrypoint is executed, the given argument `NewTokenPool` must be greater than or equal to the `TokenPool` state variable and less than or equal to the actual token reserve. See the claim `[lemma-update-token-pool-internal]` in dexter-properties.md for more details.

Assumptions

In order to prove the safety properties over arbitrary operation sequences, we had to make several assumptions on the execution environment outside the Dexter contract. Note that the safety properties we proved hold under the condition that these assumptions are satisfied throughout the operation of Dexter.

Assumptions for Tezos Smart Contract Execution

We axiomatized several primitive behaviors of the Tezos smart contract execution that are required for the safety proofs. Specifically, the assumptions include:

- The evaluation order of operations in the DFS.
- The state variables and XTZ balance of a smart contract account cannot be updated without executing the contract code.[2]
- A smart contract can emit only internal operations (i.e., operations whose source is not itself).
- The contract code is immutable. No runtime code generation nor evaluation is possible.

See the corresponding section in dexter-properties.md for more details.

Assumptions for External Contracts

We made assumptions on the behaviors of external contracts, especially the token and liquidity token contracts. These assumptions are required for the safety proofs, and thus it is important to verify that these are satisfied by the given implementation of the token and liquidity token

[2] In other blockchain systems, (e.g. Ethereum) it is possible to send currency to a smart contract account without ever executing the contract code (e.g. by designating the contract as the recipient of mining rewards or `selfdestruct` rewards).

contracts. If some of these assumptions are not satisfied for good reasons, then the proof needs to be revisited.

Here we highlight the critical assumptions. For the full list of assumptions, see the corresponding section in dexter-properties.md.

Token Contract `%transfer`

We assume that only Dexter can spend its own tokens, and no one else can. Specifically, for example, there must *not* exist any authorized users who are permitted to spend any Dexter-owned tokens in any cases. For another example, there must *not* exist a way to (even temporarily) borrow tokens from Dexter.

We also assume that the token transfer operation must update the balance before emitting continuation operations. For example, the token contract must *not* implement the so-called "pull pattern" where the transfer operation does not immediately update the balance but only allows the receiver to claim the transferred amount later as a separate transaction. Note that such a delayed balance update may lead to an exploit. (See A06 for an exploit scenario.)

Liquidity Token Contract `%mintOrBurn`

Regarding the liquidity token contract, we assume that only the `%mintOrBurn` entrypoint can alter the total liquidity supply and only Dexter is permitted to call it.

Abstract Functional Specification of Dexter Entrypoints

As mentioned earlier in the document, we slightly abstracted the functional correctness specification of each Dexter entrypoint and based the safety proofs on the abstract specifications. The corresponding section in dexter-properties.md provides the full details.

We note that we had to add an extra input validation condition for the `%removeLiquidity` specification for the invariant proof. Specifically, we added the condition that the `lqtBurned` argument must be *strictly* smaller than the `lqtTotal` state variable. This is required for the invariant of `lqtTotal` being positive. Indeed, this extra condition was already mentioned in the Dexter source code comment for the same reason. However, the current Dexter implementation does not check the condition, leaving the potential for Dexter becoming nonfunctional by mistake or corrupted admin users. It is strongly recommended to add an explicit input validation for the LqtBurned argument.

Recommendations

In this section, we make several recommendations to prevent potential usability issues, most of which are related to the deployment of Dexter exchanges.

A01: Input validation for `%removeLiquidity`

As mentioned in the Assumptions section, there exists no explicit input validation that ensures the `lqtBurned` argument to be strictly less than the `storage.lqtTotal` value, although it is implicitly required as described in the code comment. It is a better practice to have an explicit input validation instead of relying on the implicit assumption, considering the code simplicity and the small runtime overhead of such an input validation.[3]

Recommendation

Add an input validation for the `lqtBurned` argument in `%removeLiquidity`.

[3] Another trick is to have the initial `storage.lqtTotal` to be strictly greater than the initial total supply of the liquidity token. Then, it is impossible for `storage.lqtTotal` to become zero, even without any extra input validation for `lqtBurned`. However, this trick would increase the complexity of the deployment process.

A02: Uniqueness of liquidity token contracts

As mentioned in the Assumptions section, only (a single instance of) Dexter must be permitted to call `%mintOrBurn`. In other words, the liquidity token contract (or the liquidity token ID) must be *unique* for each exchange. If multiple exchanges are associated with the same liquidity token contract, then one could mint LP tokens in a cheaper pool and burn them in a more expensive pool, making a profit.

Currently there seems to exist no systematic enforcement mechanism for this uniqueness requirement in deploying a new exchange. This leaves the possibility that a new exchange could be deployed with an existing liquidity token contract that was already associated with another exchange, by mistake or maliciously.[4]

Recommendation

Have an enforcement mechanism that ensures this uniqueness requirement in deploying a new exchange.

[4] Although the current reference implementation of the liquidity token contract does not allow itself to be associated with more than one exchange because it admits only a single admin, users can come up with their own liquidity token contract implementation that may admit multiple admins.

A03: Input validation for %setLqtAddress

The `%setLqtAddress` entrypoint needs to check that `storage.lqtTotal` is *not* less than the total supply of the `lqtAddress` token. Otherwise, it may lead to a situation that certain benign users cannot redeem their liquidity later.

Scenario

1. Alice deploys a Dexter exchange with the initial `storage.lqtTotal` = 10.
2. She deploys a liquidity token contract, setting herself as initial liquidity provider by minting 20 tokens to herself, by mistake or maliciously.[5]
3. Bob adds liquidity and mints 10 liquidity tokens. Now, `storage.lqtTotal` = 20.
4. Alice burns 20 liquidity tokens, by mistake or maliciously, which drains Bob's funds from the pool.
5. Bob cannot redeem his liquidity tokens and loses his funds.

Recommendation

Have an enforcement mechanism that ensures the above requirement for the `%setLqtAddress` entrypoint.

[5] Note that, in the given reference implementation of the liquidity contract, Alice can mint the initial tokens as part of the storage initialization, even if she is not the admin.

A04: Initialization of state variables

Once the deployment and initialization of a Dexter exchange is completed, it is recommended to have `storage.lqtTotal` to be *not* (significantly) less than \sqrt(`storage.xtzPool * storage.tokenPool`). Otherwise, the minimum amount of deposits for minting non-zero liquidity tokens could be too high for normal users to afford.

Scenario

1. Suppose the initial value of the state variables are: `lqtTotal` = 1, and `xtzPool` = `tokenPool` = 10,000
2. Then one cannot mint any liquidity tokens with a deposit smaller than (10,000 xtz, 10,000 tokens).

Recommendation

Have an enforcement mechanism that ensures that the initial `storage.lqtTotal` value is not significantly small compared to the initial pool reserves.

A05: Front-running possibility during pool initialization

According to a test deployment script, origination.sh, a method for the deployment and initialization of an exchange pool is as follows:

A. Suppose Alice is the initial liquidity provider.
B. Alice deploys a Dexter exchange with setting an initial `lqtTotal` value (say *L*).
C. Alice deploys a liquidity token contract initially minting *L* liquidity tokens to herself.
D. Alice sends the initial amount of XTZ (say *X*) to the new exchange pool.
E. Alice sends the initial amount of tokens (say *T*) to the new exchange pool.
F. Alice calls the `%updateTokenPool` entrypoint.
G. Alice calls the `%setLqtAddress` entrypoint to associate the liquidity token contract with the exchange.

In the above method, it is important to atomically execute the steps D, E, and F. Otherwise, the pool reserves can be drained by front-running attacks.

Scenario

1. Suppose Alice has executed the steps A to D.
2. Alice sends a transaction to execute step E.
3. Bob front-runs to call `%tokenToXtz` with `tokensSold` = 1, before Alice's transaction for step E. This way, Bob can buy the entire XTZ reserve at only 1 token, since `storage.tokenPool` = 0 at this point.

Recommendation

Have an enforcement mechanism to ensure the pool initialization to be atomic.

A06: Verification of token and liquidity token contract implementations

As mentioned in the Assumptions section, the implementation of the token and liquidity token contracts needs to be verified to ensure that it satisfies the requirements for the safety and security of Dexter. Certain critical requirements include:

- No allowance (or borrowing) granted for the Dexter-owned tokens.
- No delayed balance update (e.g., the pull pattern) in token transfers.

Violating these requirements may lead to exploits. For example, if the pull pattern is adopted in token transfers, the following exploit is possible. A malicious user calls `%xtzToToken` and then calls `%updateTokenPool` before claiming the bought tokens. Later he claims the tokens, which makes `storage.tokenPool` to be larger than the actual token reserve, and distorts the token exchange price.

Note that the (FA1.2 or FA2) standard conformance requirement alone may not be sufficient. For example, it is unclear whether the delayed balance update violates the FA1.2 standard or not, as FA1.2 does *not* explicitly specify the atomicity requirement. Also, note that certain token implementations may intentionally violate some standard requirements for additional features required for their own purposes.

Recommendation

Clearly document that users must verify the implementation of the token and liquidity token contracts that satisfy the requirements before they associate them to a Dexter exchange to be deployed.