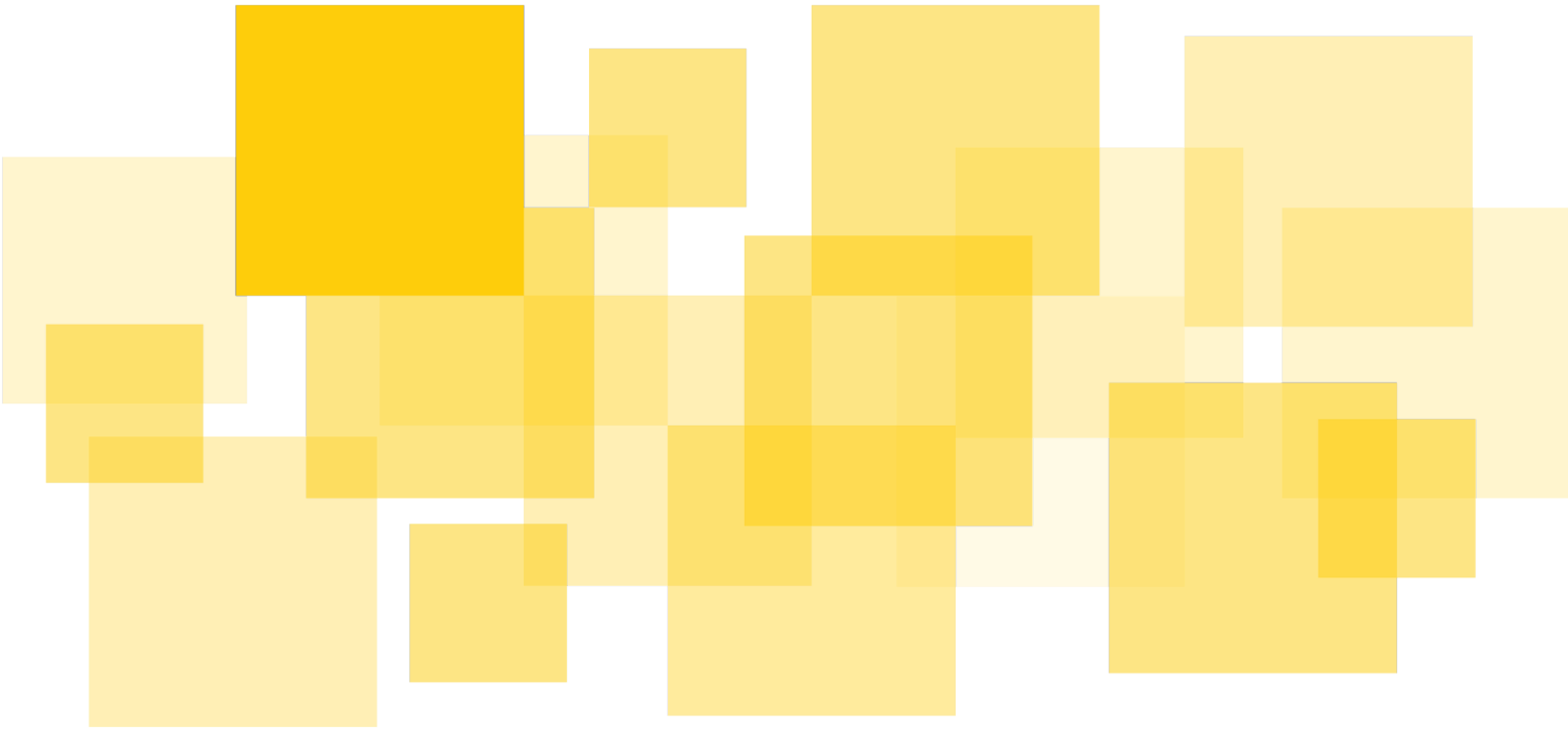
Formal Verification Report

Tezos Liquidity Baking Contract

Delivered: 2nd August 2021



Prepared for the Tezos Foundation by

runtime verification

Summary

Runtime Verification, Inc. have formally verified the Tezos Dexter Liquidity Baking (Dexter LB or Dexter, when clear from context) and Liquidity Token FA1.2 (LQT) smart contracts. The work was completed by Stephen Skeirik and Nishant Rodrigues from June 21, 2021 to July 30, 2021 in two phases. An automated formal verification of the functional correctness of each entrypoint was conducted from June 21, 2021 to July 9, 2021. Subsequently, a manual proof of some important safety properties of the system was done from July 12, 2021 to July 30, 2021.

All mechanized proofs were done in the K Framework using our existing K Michelson semantics. We note that, however, since parts of our proof have *not* been fully mechanized, they become part of our trust base. We leave the full mechanization of all proofs as future work.

Scope

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

- dexter.liquidity_baking.mligo: Liquidity baking contract source file written in LIGO
- dexter.liquidity_baking.mligo.tz: Compiled Michelson bytecode for FA1.2
- lqt_fa12.mligo: Liquidity token (LQT) contract source file written in LIGO
- lqt_fa12.mligo.tz: Compiled Michelson bytecode for the liquidity token contract

The formal verification is limited in scope within the boundary of the smart contract source 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.

We formalized baseline properties for each contract entrypoint, that specify the storage updates 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 contract source code was faithfully compiled to the Michelson bytecode, and that 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 Liquidity Baking over any sequence of arbitrary operations including non-Dexter operations. One of the most critical properties is *liquidity share value security* which states that (the geometric mean of) the XTZ and tzBTC reserves per

share *never* decreases. The share value preservation property implies that there is no way to illegally drain funds from the liquidity pool, and thus liquidity providier's funds are safe (provided that certain conditions are met, as described in the Assumptions section). Another important property is *operation safety*, i.e., do Dexter LB operations execute within reasonable time and exchange-rate bounds?

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. See our Safety Property Verification section for details on the abstraction and properties proved.

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.

K Framework

The K Framework is a rewriting-based executable semantic framework based on matching logic for defining and reasoning about systems (e.g., programming languages and type systems). K semantic specifications are based on structured configurations and rewrite rules which map configurations to new configurations. The K theorem prover is based on reachability logic (parameterized over the matching logic theory which defines the transition relation for the semantics), a logic that subsumes both Hoare logic and separation logic and which allows for reasoning inductively about any terminating behavior of a program.

K Michelson

K Michelson is a formal executable semantics for the Michelson language implemented using the K Framework. It also provides a unit-test and verification framework for Michelson programs. We gain confidence in the correctness of this semantics by cross-validating a comprehensive set of unit-tests against the reference Michelson implementation. Since the verification framework is derived from the same semantics, our confidence carries over to this as well. While the K Michelson semantics formally describes the semantics of the Michelson language, it does not yet model blockchain operations and interactions between contracts. Hence, those parts of our audit must be conducted manually.

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 lb-prove lqt-prove
```

Structure

All of the proof artifacts are included in our K-Michelson Git repository under the tests/proofs/liquidity-baking and tests/proofs/lqt directories:

- tests/proofs/liquidity-baking/lb-compiled.md
- tests/proofs/liquidity-baking/lb.md
- tests/proofs/liquidity-baking/lb-spec.md
- tests/proofs/lqt/lqt-compiled.md
- tests/proofs/lqt/lqt.md
- tests/proofs/lqt/lqt-spec.md

Where:

- *-compiled.md files contain the Michelson source code of the corresponding contract
- *.md files contain notation and definitions which enable writing the proofs compactly
- *-spec.md files contain the actual K proof scripts

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 Contract Entrypoints

Summary

In this section, we survey our methodology for verifying functional correctness of Liquidity Baking and Liquidity Token contract entrypoints. Our proof scripts include automatically verified claims for each of the entrypoints in the two contracts, and consist of two pieces:

1. The Michelson code corresponding to the contracts
2. High-level descriptions of each entrypoint derived from their 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 contract entrypoint specifies: (a) the conditions under which the transaction will succeed as well as how the storage gets updated; and (b) the conditions under which the transaction will fail (leading the Michelson VM to revert). These entrypoint descriptions as well as the proofs relating them to the Michelson code are contained in the *-spec.md files listed in the Verification Artifacts section.

Reachability Proof Claim Structure

Before we proceed, we define the basic structure of a claim. A claim has four parts with the following syntax:

```
claim Start_State => Target_State
  requires Precondition
  ensures  Postcondition
```

The parts are (in order):

1. The start state - an abstract starting state (i.e. a term with variables that describes a potentially infinite number of concrete states)
2. The target state - an abstract target state (as defined above)
3. The precondition - narrows the abstract start state by throwing away all concrete instances which do not satisfy it
4. The postcondition - narrows the abstract target state by throwing away all concrete instances which do not satisfy it

The claim is satisfiable whenever, for each concrete state *ConcS* which is an instance of our abstract start state *AbsS* that satisfies the precondition and for each execution path *P* starting from *ConcS*, we know that path *P* either:

1. reaches a state *ConcT* that is a concrete instance of our abstract target state *AbsT* such that *ConcT* satisfies the postcondition, or
2. is infinite.

In other words, the claim is ***not*** satisfied if and only if there is a finite execution path starting from our constrained abstract start state that does *not* intersect our constrained abstract target state.

Claim Configuration Syntax

The abstract start and target states are described using *configurations* in the K Framework. A configuration is an hierarchical, algebraic structure which contains all of the information about a given concrete state in a system. As an example, the configuration below describes a Michelson interpreter which is about to execute the `ADD` instruction when the input stack contains just two values: the integers 1 and 2.

```
<k> ADD </k>
<stack> [ int 1 ] ; [ int 2 ] ; .Stack </stack>
```

The following claim describes how the above configuration executes:

```
<k> ADD => . </k>
<stack> [ int 1 ] ; [ int 2 ] ; .Stack => [ int 3 ] ; .Stack </stack>
```

That is, the current continuation, the `ADD` instruction, evaluates to the empty continuation, and the two stack values evaluate to the single stack value 3. Each piece of the configuration can update independently. Any cell of a configuration (e.g. `<k>`) that does not contain an arrow (`=>`) denotes that the abstract start and target states have an identical value for that particular cell.

Example

As an example, we illustrate a part of the functional correctness proof claim for the `%getBalance` entrypoint of `lqt_fa12.mligo`. The code in the diagram below defines the claim which specifies half of the positive case (i.e., the contract succeeds and generates a callback which contains the balance).

```
claim

  <k> #runProof(GetBalanceParams(Address, Callback)) => .K ... <k>
  <stack> .Stack <stack>
  <myamount> #Mutez(Amount) </myamount>
  <nonce> #Nonce(Nonce => Nonce +Int 1) </nonce>

  <tokens> Tokens <tokens>
  <operations> _
            => [ Transfer_tokens
                 {Tokens[Address]}:>Int
                 #Mutez(0)
                 Callback
                 Nonce ]
            ;; .InternalList
  </operations>

requires
  Amount ==Int 0 andBool Address in_keys(Tokens)
```

As is usual when writing a proof claim, we use both the logical theory which is defined by our system under evaluation (the *system specification*) as well as a more general logical theory which allows us to define properties of interest over our system (the *property specification*). In this case, the system specification defines the Michelson semantics while the property specifications describe how particular contract entrypoints transform the Michelson contract storage and internal operations queue.

In the above example, `#runProof` is a proof-specific function that:

1. Consumes an abstract representation of the LQT contract parameters and storage values;
2. Constructs a proper initial stack for the LQT contract based values from (1);
3. Executes the LQT code from the given starting stack to produce the final stack value;
4. Updates the operations stack and contract storage based on the values from (3).

So, to summarize, what this claim says is: when the LQT contract receives a valid `input` on its `%getBalance` entrypoint, if the amount paid to the contract is 0 *and* there exists a map entry keyed by Address in the Tokens map, then the whole state is unchanged *except* for:

1. the current continuation (e.g. the `<k>` cell) becomes empty,
2. the nonce is incremented once for the emitted internal operation,
3. a single operation is emitted onto the operation stack.

Conclusion

Given that we believe our proof scripts are fairly declarative, as in the example above, we do not summarize them any further here and instead refer the reader to our proof claim descriptions located in the K Michelson source tree at:

For the Liquidity Baking contract:

https://github.com/runtimeverification/michelson-semantics/blob/master/tests/proofs/liquidity-baking/lb-spec.md

For the LQT contract:

https://github.com/runtimeverification/michelson-semantics/blob/master/tests/proofs/lqt/lqt-spec.md

Note that, unlike Coq/Isabelle/etc, proofs using the K Framework are typically done in a one-shot, fully automated fashion. This means that, when we use K, we don't have proof scripts in the usual sense, i.e., a list of proof tactics that takes us from A to B. Rather, we list our proof claims and let the proof engine discharge them automatically for us. Currently, we do not have a fully-fledged proof checker, but such a proof checker is under development. By virtue of this, our proofs are typically quite short.

Safety Property Verification

Summary

Building on our functional correctness proofs, we state and manually prove two important safety properties:

1. *Liquidity share value security*: the exchange value of a liquidity share never decreases. Formally, when we speak of the *exchange value* of a liquidity share, we mean:

$$\text{exchange value} := \frac{\sqrt{\textit{the XTZ reserve} * \textit{the token reserve}}}{\textit{the total liquidity shares}}$$

   Note that the exchange value of a share is different from its monetary value (an example of the *impermanent loss* problem).

2. *Operation safety*: each operation executes within worst-case time and exchange-rate bounds. Formally, we mean that each operation carries arguments which bound:

   - the time before which they are permitted to execute (deadline);
   - the minimum value they receive in exchange for assets sent (either XTZ, tzBTC, or LQT);
   - the maximum value of additional assets they must send in exchange for receiving some asset (tzBTC).

   The last two bullets are examples of *slippage protection*, which protects liquidity providers (LPs) and traders from too much variation in exchange rate due to the asynchronous nature of transaction execution.

Note that the liquidity share value security property subsumes safety properties such as:

- 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.

We proved that the share price preservation property holds over any sequences of arbitrary operations. The proof is based on:

1. an *abstracted* Tezos blockchain environment which we describe below;
2. several key lemmas which we proved.

We address each of these below. For full details of the proofs, see:

https://github.com/runtimeverification/michelson-semantics/blob/master/tests/proofs/liquidity-baking/lb-properties.md

Abstract Blockchain Environment

K Michelson does not yet provide a formal semantics for the Tezos blockchain environment. This means we must develop an abstracted blockchain over which we manually prove our high-level properties. Note that, in any case, we must abstract some details of the blockchain for the proof to be useful, e.g., the proof cannot depend on a concrete snapshot of the blockchain at some particular moment. That is, the proof must be parameterized over all possible blockchain states that correspond to a possible evolution of the initial state in which Dexter LB was properly deployed. We give the details of this abstract environment below.

Global Assumptions. Globally, we assume several points about how Tezos operates:

- The evaluation order of operations is DFS.
- The state variables and XTZ balance of a smart contract account cannot be updated without executing the contract code.[1]
- A contract may only emit internal operations (i.e., operations whose source is not itself).
- All contract code is immutable.
- No runtime code execution is possible except where a contract entrypoint executes a supplied `lambda` (but since this never occurs in our modelled contract interfaces, we can safely ignore this case).

Contract Abstractions. Our abstract environment does *not* contain full details of all Tezos contracts on the chain. In particular, we *only* consider the following contracts:

- Dexter LB
- LQT
- tzBTC
- Other Dexter V2 contracts called via Dexter LB's `%tokenToToken` entrypoint

Operation Abstractions. We *only* consider operations sent to or from the Dexter LB, LQT, or tzBTC contracts.

Behavior Abstractions. We model the behavior of the four contracts listed above abstractly, as follows:

[1] 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).

- **Dexter LB & LQT.** Their behavior is given by high-level K specifications which we formally verified are consistent with their Michelson source code (see more details in Functional Correctness of Contract Entrypoints).
- **tzBTC.** Its behavior is assumed to conform to the FA1.2 specification in a generic way but with some additional restrictions listed below (see the Additional Assumptions); note that, in some sense, this is the best we can do, given that tzBTC contract is *upgradable*, such that its behavior is not fully specified by a fixed Michelson source code string (i.e., its execution depends on `lambda`'s in its storage).
- **Other Dexter V2 contracts.** Their behavior is given by high-level K specifications which we previously formally verified are consistent with the Dexter V2 Michelson (see: https://github.com/runtimeverification/publications/blob/main/reports/smart-contracts/Tezos-Dexter.pdf).

Storage Abstractions. We ignore updates to storage cells which are irrelevant to proving the safety property at hand. More precisely, we ignore updates to:

- the allowances map in both the tzBTC and LQT contracts;
- the LQT balances of all addresses in the LQT contract;
- the tzBTC balances of all addresses in the tzBTC contract *except* Dexter LB's balance;
- the storage value of all contracts on the chain *except* Dexter LB, LQT, and tzBTC.

We further assume that all contract storages were properly initialized:

- Dexter LB's contract has been initialized to contain the correct addresses of the tzBTC and LQT contracts and that its initial tokenPool and lqtSupply variables correspond to its tzBTC balance and the total supply of LQT in the LQT contract;
- LQT's initial storage contains Dexter LB as the admin and LQT's total supply variable corresponds to the total amount of all LQT minted belonging to all users at contract creation.

Additional Assumptions. We make a small number of additional assumptions about the behavior of our smart contracts listed above.

- **Dexter tzBTC 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. There must *not* exist a way to (even temporarily) borrow tokens from Dexter.
- **tzBTC `%transfer` Behavior.** 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.

Key Lemmas

Our main safety properties follow from several lemmas that we proved:

- each Dexter state variable eventually is consistent with the underlying blockchain state value it represents;
- The total supply of liquidity tokens equals the totalSupply variable in the LQT contract storage;
- Only the mintOrBurn entrypoint can update the LQT contract's total supply of liquidity token and only Dexter LB may call it.

Each lemma was proved by the induction on sequences of operations and case analysis over different types of operations. We divide these lemmas into two categories based on the contract they describe.

Dexter Contract Lemmas

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 tzBTC reserve size, and the total number of liquidity shares (LQT), 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 re-entered during the execution of the current continuation operations, and this re-entrance can be repeated, the gap between the state variables and the actual reserves/shares is unbounded.

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.

LQT Contract Lemmas

To prove that the `LqtTotal` state variable reflects the actual liquidity shares we need two important lemmas about how the number of liquidity shares may change. The first states that only the Dexter contract may update the total number of shares, and only through using the `%mintOrBurn` entrypoint. This requires a further lemma — the total number of shares recorded in the LQT contract's storage reflects the actual value i.e. the sum of the shares owned by each shareholder. The second is a liveness property — when the Dexter contract calls the `%mintOrBurn` entrypoint (meeting certain preconditions) the total number of shares is actually updated.

Recommendations

In this section, we make several recommendations to prevent potential usability issues. Due to the similarity between this contract and the Dexter V2 contract, we will refer the reader to our Dexter V2 contract audit report, available at the link below:

https://github.com/runtimeverification/publications/blob/main/reports/smart-contracts/Tezos-Dexter.pdf

In that report, recommendations A01 and A04-A05 still apply. Additionally, recommendation A06 as it pertains to the tzBTC contract still applies.

A01: Unintended Mutez Overflow in token_to_xtz

When calculating the amount of XTZ purchased (less the portion burned), the code here is used (reproduced below for the reader's convenience):

```
let xtz_bought = natural_to_mutez
  (((tokensSold * fee * (mutez_to_natural storage.xtzPool)) /
    (storage.tokenPool * 1000n + (tokensSold * fee))))
in
let xtz_bought_net_burn =
  let bought = (xtz_bought * 999n) / 1000n in
    if bought < minXtzBought
      then (failwith error_XTZ_BOUGHT_... : tez)
      else bought
```

Due to the fact that the variable `xtz_bought` has type `mutez`, the type of the variable `bought` (and the intermediate values used to produce it), also have type `mutez`.

Thus, even though we know that:

`bought = ((xtz_bought * 999n) / 1000n) <= xtz_bought`

The intermediate value `xtz_bought * 999n` may overflow, leading to the strange result that we cannot convert tzBTC to tez if the price after conversion is within ~1000 times of the maximum representable `mutez` value.

As far as we can tell, this issue does not lead to privilege escalation in the sense that the contract would perform an *unintended* behavior. Instead, this issue has the potential to prevent a user from executing an *intended* behavior.

Recommendation

To avoid this issue, we recommend for all calculations which produce a `mutez` type that:

a. All `mutez` inputs are converted into number types before the calculation starts
b. The final result value is casted back into the `mutez` type after the calculation is complete

In this specific case, one possible solution is to apply a `natural_to_mutez` conversion to the variable `xtz_bought` in the calculation of the `bought` variable.

Status

Client acknowledged the issue. Both client and we agree that this issue is not a security issue but rather a usability issue. Client further noted, and we agree, that:

1. With the current codebase, it is still possible to execute large trades from tzBTC to tez using the `token_to_xtz` entrypoint if the trade is broken up into multiple transactions.
2. The issue may be resolved in a forthcoming protocol amendment which would change the representation of the `mutez` type internally from fixed to arbitrary precision arithmetic such that overflow and underflow would become impossible. In such a case, our recommendation to convert `mutez` to a number type for intermediate calculations would no longer apply.

A02: Unintended Mutez Overflows in token_to_token

When calculating the amount of XTZ purchased (less the portion burned), the code here is used (reproduced below for the reader's convenience):

```
let xtz_bought =
  (tokensSold * fee * storage.xtzPool) /
  (storage.tokenPool * 1000n + (tokensSold * fee))
in
let xtz_bought_net_burn = (xtz_bought * 999n) / 1000n
```

This code block contains two possible `mutez` overflows:

1. Since the value `storage.xtzPool` has type `mutez`, by the typing rules of the multiplication operation, the intermediate value:

   `(tokensSold * fee * storage.xtzPool)`

   also has type `mutez`. This means that if:

   `tokensSold * storage.xtzPool ~= mutez_max / 1000`

   the subcomputation will overflow.

2. By the typing rules of the multiplication and division operations, the variable `xtz_bought` has type `mutez`. This means that the subcalculation `xtz_bought * 999n` will overflow whenever:

   `xtz_bought ~= mutez_max / 1000`

   even though it is guaranteed that:

   `xtz_bought_net_burn <= xtz_bought`.

As in A01, as far as we can tell, this issue does not lead to privilege escalation in the sense that the contract would perform an *unintended* behavior. Instead, this issue has the potential to prevent a user from executing an *intended* behavior.

Recommendation

Our overall recommendation is the same as in finding A01, with the slight difference being that the `mutez_to_natural` conversion function should be applied to both `storage.xtzPool` and

the `xtz_bought` values in this case. This will require an extra `natural_to_mutez` at the end of the calculation of `xtz_bought` to satisfy the typing rules.

Status

Client acknowledged the issue. Both client and we agree that this issue is not a security issue but rather a usability issue. As noted above, a forthcoming protocol amendment may resolve this issue entirely.

A03: token_to_token Unusable If Contract Has Too Much XTZ

Using the mutez overflow from finding A02, we note that if the `storage.xtzPool` value is large, then the mutez calculations in the `token_to_token` entrypoint will overflow. In particular, this means that this entrypoint cannot be used in that case, i.e., when the liquidity pool has lots of XTZ through donations via `default` or else through liquidity contributions via `add_liquidity`. Of course, we can still manually call `token_to_xtz` and then `xtz_to_token` from a different Dexter contract.

Recommendation

Our recommendation is the same as in finding A02.

Status

Client acknowledged the issue. Both client and we agree that this issue is not a security issue but rather a usability issue. As noted above, a forthcoming protocol amendment may resolve this issue entirely.