iCon: Automated Verification of Inter-Transaction Properties in Tezos Smart Contracts with Unknowns

Anonymous Authors

***Abstract*—Smart contracts play a critical role in blockchain applications, managing vast amounts of valuable assets. However, they are often vulnerable to attacks due to the inherent difficulties in modifying their code once deployed. Existing security analysis tools and verifiers primarily focus on single-contract verification, while many real-world blockchain applications involve multiple contracts and transactions. In this paper, we introduce an automated verifier, iCon, for inter-transaction properties of smart contracts on the Tezos blockchain platform. iCon is based on our program logic, which verifies inter-transaction properties in the presence of both known and unknown contracts. We present an abstraction technique for unknown contracts and propose a proof technique to ensure that an inter-transaction property holds for any existence of unknown contracts. The proof technique supports the correctness of our verification approach. We have implemented iCon on top of the Why3 verification framework, demonstrating its effectiveness through several case studies, including the decentralized exchange service Dexter2, of which the previous version had a flaw reported in its implementation.**

I. Introduction

As blockchain technology continues to serve as a foundation for various distributed applications, nearly all blockchain platforms now support the execution of programs—called *smart contracts* or simply *contracts*—deployed and executed on a blockchain. Since the first proposal by [1] and the introduction by Bitcoin [2], numerous blockchains, such as Ethereum [3], [4] and Tezos [5], have incorporated smart contract functionality, allowing for the development of complex programs using Turing-complete languages. These applications span various uses, from casual games to decentralized finance (DeFi) platforms that manage substantial amounts of valuable cryptoassets.

Typically, a smart contract is associated with a blockchain account and invoked when the account receives cryptoassets, called *tokens*, in a transaction from another account. The sender account can also pass a parameter along with the transaction, which is then used for computation by the contract associated with the receiving account. Furthermore, a smart contract can save values on the blockchain, making it stateful.

As with any programming platform, bugs are inevitable also in smart contracts. Smart contracts have been exposed to various attacks due to the amount of assets they manage and the inherent difficulty of modifying their code once deployed, which makes them attractive targets for attackers. As a result, the cryptoasset market loses several billion U.S. dollars each year, with many of these losses resulting from attacks on DeFi applications exploiting smart contract vulnerabilities.[1]

[1]https://go.chainalysis.com/2023-Crypto-Crime-Report.html

Numerous security analysis tools [6] and automated verifiers [7]–[14] for smart contracts have been proposed to address this issue. However, many of them primarily focus on verifying single contracts [11]–[14], while many practical blockchain applications involve multiple contracts and transactions. Especially for the Tezos platform, only a few properties involving multiple contracts have been proven using the Coq proof assistant system [15], which requires a lot of manual effort to ensure safety.

Our goal is to develop an automated verifier for *inter-transaction properties*, specifically targeting smart contracts on the Tezos blockchain platform [5]. The main challenge is that their safety depends not only on the *known* contracts constituting the application but also on *unknown* contracts not incorporated into the application; by taking an account address as a parameter and then sending tokens to the account, a contract can execute code not part of the application. Additionally, such code might not be deployed at the verification time.

To this end, we design a program logic for verifying inter-transaction properties in the presence of unknown contracts as the foundation of our approach. Although this program logic does not directly apply to verification, we develop a verification technique and show its correctness using the program logic. Our technique takes user-annotated known contracts and the invariants to hold for unknown contracts as input, verifying the given application based on our program logic.

We have implemented our verification method on top of the Why3 verification framework [16]. The effectiveness of our verifier, called iCon, was demonstrated through several case studies, including the decentralized exchange service Dexter2, of which the previous version had a flaw in its implementation [17], [18].

Our contributions can be summarized as follows:

- We propose a verification technique for the inter-transaction properties of Tezos smart contracts. To this end, we formalize the inter-transaction semantics of Tezos, define a program logic that verifies inter-transaction properties in the presence of unknown contracts, and introduce an abstraction technique for unknown contracts.
- We implement our technique on top of the verification platform Why3, allowing users to provide the implementation and specification for known contracts and the model of unknown contracts using an extension of WhyML—the functional programming language provided by Why3.

- We demonstrate the effectiveness of our verifier iCon through several interesting case studies, including the decentralized exchange service Dexter2.

The outline of this paper is as follows: Section II explains the execution model of Tezos smart contracts; Section III presents an overview of our verification approach and the challenges associated with it; Section IV describes our implementation, iCon; Section V introduces the case studies we conducted; Section VI formalizes the inter-transaction semantics of Tezos, defines the program logic for our verification, and proves the soundness of the program logic; Section VII discusses related work; and Section VIII concludes.

A. Limitation

The current implementation has several limitations in both theoretical and engineering aspects. The major limitations are as follows.

- Our implementation assumes that a known contract does not deploy new contracts. We need to extend our theoretical foundation to handle such contracts, which is left as future work.
- A known contract can call other contracts only up to a fixed number of times. This limitation simplifies the verification process. We believe this is not a substantial limitation, as most contracts on the Tezos blockchain satisfy this limitation.

II. Execution model of Tezos smart contracts

Like many other blockchain platforms, the Tezos blockchain platform is a platform for managing a distributed ledger that stores various data associated with accounts. The state of this database is updated by submitting data—called *operations*—to the Tezos network, which is then incorporated into the blockchain.

Accounts on the Tezos platform are categorized into *implicit accounts* and *smart contract accounts*. An implicit account is owned by a user identified by its public key; it is not associated with any code but can spontaneously submit an operation and be the starting point of a chain of transactions. A smart contract account is associated with a program called a *smart contract* or simply a *contract*. This program is written in Michelson, the virtual machine language of the Tezos blockchain, and executed when the smart contract account to which the code is associated receives *XTZ*, the native cryptocurrency of Tezos. A smart contract can record values on the *storage* field of the associated account; therefore, a smart contract can be stateful.

There are three primary types of operations: *transactions*, *originations*, and *delegations*. A transaction operation transfers tokens from one account to another and can result in the invocation of a contract. An origination operation deploys a new smart contract account. A delegation operation delegates the right to participate in block baking—a part of Tezos's Proof of Stake consensus mechanism—from one stakeholder to another. These operations can be submitted by a contract, potentially leading to a chain of contract invocations.

Algorithm 1: Pseudocode describing the effect on an account that receives *amount* XTZ

```
1  // self denotes the account receiving XTZ
2  self.balance ← self.balance + amount;
3  // If the account is a smart contract
4  if ∃ self.code then
5      // Run michelson VM
6      let o⃗, s′ = mVM(self.code, p, self.storage);
7      self.storage ← s′;
8      // Process the list from head
9      foreach o in o⃗ do
10         switch o do
11             // Transaction operation
12             case Xfer(v, m, a) do
13                 self.balance ← self.balance - m;
14                 send(v,m,a);
15             end
16             // Origination operation
17             case Orig(c, v, m, a) do
18                 self.balance ← self.balance - m;
19                 create(c,v,m,a);
20             end
21             // Delegation operation
22             case Sdel(a) do
23                 self.delegate ← a;
24             end
25         end
26     end
27 end
```

Algorithm 1 describes how the blockchain works when an account receives tokens. Line 2 changes the balance *self.balance* of the account by the transferred amount. Then, Line 4 decides whether the account is an implicit account or a smart-contract account by checking whether the account has a program represented by *self.code*. If the account is a smart-contract account, the contract associated with the account is executed (Lines 6 to 26), which proceeds as follows.

- The contract code *self.code* is passed to the Michelson virtual machine, mVM, with the contract parameter *p* bundled in the transaction operation, and the current storage value *self.storage* (Line 6). mVM does not modify the block state or invoke another contract at this instance; it only calculates values $\vec{o} = [o_1, \ldots, o_n]$ representing the operations executed after the current contract execution and s', the new storage value. $\vec{o}$ is referred to as *internal operations*.
- The storage value is updated from s to s' (Line 7).
- The internal operations $o_1, \ldots, o_n$ are executed in head-to-tail order (Lines 9 to 26). A transaction operation, denoted by $Xfer(v, m, a)$, consists of a contract parameter v, the amount of tokens sent m, and a destination address a. Following these parameters, the account balance is reduced (Line 13), and then the tokens are sent to the

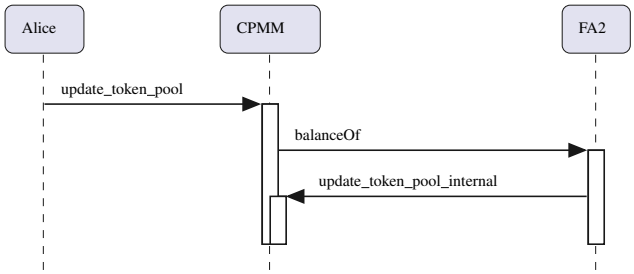

Fig. 1. updateTokenPool

destination account along with the contract parameter, as indicated by the `send` call (Line 14), resulting in recursive invocations of Algorithm 1 for the account at address a. The origination operation *Orig* and the delegate operation *Sdel* are processed in accordance with the intuition explained above.

A series of executions initiated by an implicit account may fail for several reasons, including an exception thrown by the contract code or a lack of balance (cf. Line 13). In such cases, only the fact of failure is recorded, and all changes made during the executions are discarded.

In the following, we illustrate the inter-transaction behavior involving multiple contracts using a sequence diagram exemplified by Fig. 1. In a sequence diagram, each participant denotes an account, each message denotes a transaction operation, where the label denotes the entrypoint being invoked, and each execution specification (a thin box on a lifeline) corresponds to the execution of Algorithm 1. Note that the entrypoint mechanism uses a tagged value for a contract parameter. So, no extension is required to what we have explained. We do not explain what the inter-transaction illustrated in Fig. 1 is for, but we can notice several things: an implicit account owned by Alice starts a transaction, calling `update_token_pool` entrypoint of CPMM; CPMM executes its code and issues one internal operation calling FA2; and FA2 callbacks CPMM, where we overlap the execution specification activated by the callback as intended.

III. Verification Approach

Our verification methodology is grounded in a *software-contract*–based approach [19]. This approach entails a rigorous examination of blockchain states before and after each contract invocation, ensuring compliance with predefined pre- and post-conditions. Importantly, if contract execution fails, all modifications made by the contract are reverted. Therefore, our analysis is confined to scenarios involving successful contract executions.

This straightforward approach, however, encounters two primary challenges:

1) The dynamic composition of internal operations within contracts obscures the potential contract calls, necessitating a detailed analysis to reason the overall contract behavior.
2) The involvement of unknown contracts, especially third-party contracts with unspecified behaviors, makes it impractical to assume complete knowledge of all contracts engaged in an invocation.

To address these challenges, we categorize blockchain contracts into two types: *known* and *unknown contracts*. Known contracts are the focus of our verification and therefore their state changes are of our interest. Their code is accessible at verification time. Conversely, unknown contracts may not have their code available at the time of verification and are treated as a collective entity with a single pre-condition and post-condition for their behavior. It is noteworthy that some unknown contracts may not even be implemented when the verification is conducted.

For known contracts, we conduct sequential program reasoning to verify that contract execution outlined in Algorithm 1 conforms to the specified pre- and post-conditions. This involves unfolding the foreach loop in Algorithm 1 into a sequential process, based on the assumption of a fixed length for the internal operation list. A limitation of iCon is its restriction to known contracts with a predefined upper limit for the number of issued internal operations. The loop reasoning becomes straightforward as we can deduce the pre- and post-conditions of `send` from the given conditions.[2] The remaining challenge lies in reasoning about the execution of mVM. Here, we leverage the functional property of the code, which defines the relationship between input and output of mVM relative to the code. Various existing tools [11]–[14] are utilized to verify the code's compliance with its functional property and confirm the correctness of the assumed upper bound.

For unknown contracts, we model their behavior using *history invariants*, a concept originally proposed for verifying object-oriented programs [20], [21]. In our context, a history invariant describes the evolution of blockchain states, capturing invariant relations before and after a series of operations. Our adaptation of this concept involves proving the effect of the foreach loop in Algorithm 1 for any length of internal operation list through induction. This proof is feasible when the pre- and post-conditions of unknown contracts form a history invariant, a modeling technique also used in Ethereum smart contract verifiers [9], [10]. Fortunately, the unknown contract's state changes do not impact the properties being verified, allowing us to exclude the unknown contract's code from our analysis. This approach facilitates the verification of properties without access to the unknown contract's code.

IV. iCon: inter-transaction verifier

We have developed iCon, an inter-transaction verifier for the Tezos blockchain platform. This verifier follows the approach outlined in Section III and is implemented as a plugin for the Why3 platform. On receiving an input file with necessary verification information, the plugin generates WhyML code representing a proof of the verification. The correctness of

[2]As previously mentioned, reasoning about the *Orig* case is a future work area, while *Sdel* is considered a null operation in our context.

```
1  scope Unknown
2    (* Pre- and post-condition for unknowns *)
3    predicate pre (c : ctx) = ...
4    predicate post (c : ctx) (c' : ctx) = ...
5  end
6
7  scope Contract_i
8    (* Type of storage *)
9    type storage = ...
10
11   (* Pre- and post-condition of contract *)
12   predicate pre (c : ctx) = ...
13   predicate post (st : step) (gp : gparam)
14     (c : ctx) (c' : ctx) = ...
15
16   (* Upper bound of # of internal operations *)
17   let upper_ops = n
18
19   scope Spec
20     (* Functional property of each entrypoint *)
21     predicate entrypoint_i (st : step)
22       (p_1 : t_1) ... (p_n : t_n)
23       (s : storage) (o : list operation)
24       (s' : storage) = ...
25   end
26 end
```

Fig. 2. Skeleton of the iCon Input File

this proof is established through the automatic or interactive discharge of verification conditions using Why3's functionalities.

The input file for iCon, structured in WhyML code, has a fixed format as depicted in Fig. 2. Its top-level consists of `scope` declarations detailing each contract's information. These include storage type definition (Line 9), pre- and post-conditions (Lines 12 to 14), an upper bound of the number of internal operations (Line 17), and functional properties for each entry point (Lines 19 to 25).

Pre-conditions are predicates over blockchain states, represented by the `ctx` type. The definition of `ctx` varies based on the given contracts, forming a record type whose fields consist of the balance and storage values of these contracts. This design implies that one can the state of only the known contracts.

Post-conditions, on the other hand, are predicates over the calling context (represented by the `step` type), contract parameters (the `gparam` type), and the blockchain states before and after contract execution (denoted by c and c' in the figure). The `gparam` type represents a tagged value different from raw entrypoint parameters. For example, `Gp'0default'0int 10` represents a tagged value passed to a contract to invoke the `default` entrypoint with the parameter `10`. We use the shorthand `Gp'`*contract*`'`*entrypoint* for convenience, allowing `Gp'C'default` to be used in place of `Gp'0default'0int` when a contract `C` has a `default` entrypoint with an `int` parameter type.

The upper bound of internal operations is defined as a natural number, which is currently assigned manually.

Each entrypoint's functional property is a predicate over the calling context information, contract parameters, original storage value, the internal operations issued, and the updated storage value.

V. Case Studies

As a case study, we have examined three applications: Boomerang, Auction, and Dexter2, to evaluate the usability and effectiveness of our tool. The first two applications come from Tezos and Ethereum tutorials, whereas Dexter2 is derived from an industrial context. Here, we provide a concise overview of Boomerang and Auction, followed by a detailed examination of Dexter2.

A. Overview of Verification of Boomerang and Auction

Boomerang is a popular tutorial example of Tezos smart contracts. It operates by receiving tokens and then returning these tokens to the sender. Concurrently, it tracks the cumulative amount of the tokens received in its storage. Using iCon, we successfully verified that Boomerang's balance remains constant while its storage value monotonically increases.

The Auction contract, derived from a Solidity tutorial[3], facilitates a straightforward open auction where participants can place bids using Tezos tokens. The Auction contract retains these bids, allowing participants to withdraw their bids at any point, barring the highest bid. Once the auction concludes, the highest bid is awarded to the beneficiary. We verified that the Auction contract's balance is always not less than the aggregate of all bids, ensuring that each participant can reclaim their deposit.

B. Verification of Dexter2

Dexter2[4] is designed as a decentralized exchange platform where clients can trade XTZ against a variety of standardized tokens, including ones that follow FA1.2[5] and FA2[6]. This service is structured as a composite of simpler units, each handling the exchange between XTZ and a specific standardized token. Additionally, Dexter2 enables the exchange between two standardized tokens through the automated execution of two XTZ-denominated exchange units.

Each Dexter2 unit comprises three contracts: CPMM, LQT, and a targeted token exchange contract. The CPMM acts as the primary interface for the exchange. LQT serves a critical function beyond mere exchange——it manages the liquidity, specifically the reserves of XTZ and tokens that CPMM possesses. Dexter2 incentivizes users to contribute to these reserves, thereby facilitating a non-custodial service. LQT adheres to the FA1.2 standard, representing the reserves each user contributes. These tokens can be minted or burned via the exclusive `mintOrBurn` entrypoint, accessible only by CPMM. As users contribute or withdraw reserves, corresponding LQT tokens are minted or burned, respectively. Fig. 3 depicts this process of liquidity provision and withdrawal by a user, exemplified by Alice.

[3]https://github.com/ethereum/solidity/blob/efed3b2355b0840cfc884e383211f0a988937367/docs/examples/blind-auction.rst

[4]https://gitlab.com/dexter2tz/dexter2tz/

[5]https://gitlab.com/tezos/tzip/-/blob/master/proposals/tzip-7/tzip-7.md

[6]https://gitlab.com/tezos/tzip/-/blob/master/proposals/tzip-12/tzip-12.md

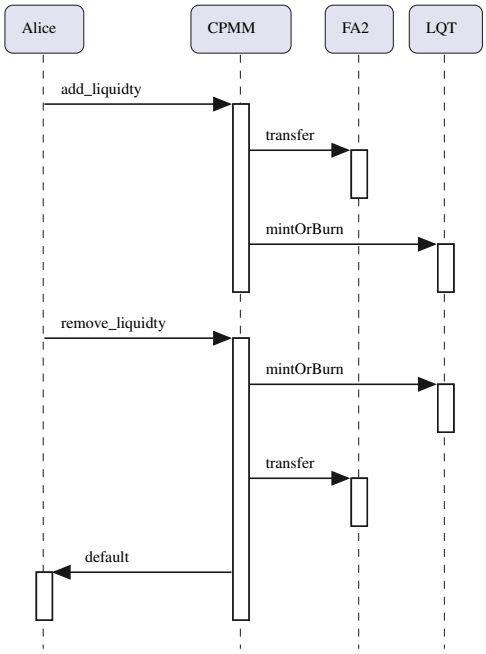

Fig. 3. Interaction of providing and withdrawing reserves

For this case study, our focus is on the property of liquidity management. Hence, we omit detailed discussions of entrypoints related to the exchange functionality. Crucially, these entrypoints do not alter `lqt_total`, a key value in CPMM's storage involved in our verification, nor do they invoke LQT.

1) Invariant of Dexter2 Exchange: We focus on a crucial invariant within the Dexter2 exchange. Specifically, we have verified that the value of `lqt_total` in CPMM's storage consistently equals the value of `total_supply` in LQT's storage. This equivalence is vital as `total_supply` records the total amount of LQT tokens in circulation.

The importance of this invariant is tied to operational efficiency. During the process where a user either provides or withdraws reserves, CPMM is responsible for calculating the amount of LQT tokens to be minted or burned. This calculation depends on the existing total supply of LQT tokens. However, due to the Tezos smart contract execution model, directly and efficiently accessing the total LQT token count during these calculations is difficult. Thus, keeping `lqt_total` in sync with `total_supply` becomes essential for the smooth functioning of CPMM.

Informally, the invariant is maintained as follows. The total LQT supply is modified exclusively through the `mintOrBurn`, which is under the control of the LQT's administrator, assumed to be CPMM. Exchange-related entrypoints do not alter `lqt_total` or interact with LQT directly. Therefore, the primary entrypoints of concern for maintaining this invariant are `add_liquidity` and `remove_liquidity`, as depicted in Figure 3. These entrypoints adjust `lqt_total`, and the subsequent `mintOrBurn` operation alters `total_supply` equivalently, thereby preserving the invariant. Figure 4 represents the functional property of the `add_liquidity` entrypoint.

```
1  predicate add_liquidity (st : step)
2   (p1 : address) (p2 : nat) (p3 : nat)
3   (p4 : timestamp) (s : storage)
4   (op : list operation) (s' : storage) =
5    s.lqt_address <> null_addr /\
6    s'.lqt_address = s.lqt_address /\
7    let lqt_minted = s'.lqt_total - s.lqt_total in
8    match op with
9    | Cons (Xfer (Gp'Unknown'transfer _) _ _)
10     (Cons (Xfer (Gp'Lqt'mintOrBurn q _) _ lqc)
11       Nil) ->
12     q = lqt_minted /\
13     lqc = s.lqt_address
14   | _ -> False
15   end
```

Fig. 4. Specification for the `add_liquidity` Entrypoint

Ideally, this invariant would serve as the pre- and post-condition for contracts within Dexter2. However, this is not feasible due to two main reasons: the need for additional invariants (such as ensuring LQT's admin remains to be CPMM) and this invariant is temporarily broken at certain stages of pre- and post-condition checks. For example, during the `add_liquidity` operation, `lqt_total` is updated before LQT is called, leading to a temporary mismatch with `total_supply`. This aspect emphasizes the complexity and dynamism of maintaining invariants within smart contracts like Dexter2.

2) Pre- and post-condition for Dexter2 contracts: We introduce the pre- and post-conditions used in the verification of the Dexter2 contracts. Although the conditions introduced here are simplified versions, they hold in a stable state following the initialization of the Dexter2 contracts, such as setting the LQT's administrator to CPMM. It is important to note that due to mutual dependencies between LQT and CPMM, which are determined during deployment, the configuration cannot be hardcoded[7].

The rationale behind presenting the simplified conditions is twofold: firstly, the more stringent conditions are considerably intricate, and secondly, even these simplified versions warrant careful examination due to their nuanced nature.

a) CPMM contract: Fig. 5 delineates the pre- and post-conditions for the CPMM contract. The first three equations in both `pre` and `post` (Lines 2 to 5 and Lines 9 to 12) specify invariants in the traditional sense, ensuring consistency before and after CPMM invocation. In contrast, the last equation in `post` (Lines 13 to 16) introduces a history invariant, crucial for elucidating the key invariant, and highlights a subtly different aspect from the key invariant.

The first two equations specify the exchange service configuration, ensuring CPMM's liquidity token contract address (`lqt_address`) is set to LQT's address (`Lqt.addr`), and setting LQT's administrator to CPMM.

The third equation's presence, while seemingly extraneous, is essential. It asserts the equivalence of the cached

[7]The addresses of LQT and CPMM are interdependent and assigned during deployment, necessitating dynamic configuration.

```
predicate pre (c : ctx) =
  c.cpmm_storage.Cpmm.lqt_address = Lqt.addr ∧
  c.lqt_storage.Lqt.admin = Cpmm.addr ∧
  c.lqt_storage.Lqt.total_supply
    = sum_of c.lqt_storage.Lqt.tokens

predicate post (st : step) (gp : gparam)
                (c : ctx) (c' : ctx) =
  c'.cpmm_storage.Cpmm.lqt_address = Lqt.addr ∧
  c'.lqt_storage.Lqt.admin = Cpmm.addr ∧
  c'.lqt_storage.Lqt.total_supply
    = sum_of c'.lqt_storage.Lqt.tokens ∧
  c'.cpmm_storage.Cpmm.lqt_total
   - c'.lqt_storage.Lqt.total_supply
  = c.cpmm_storage.Cpmm.lqt_total
     - c.lqt_storage.Lqt.total_supply
```

Fig. 5. Pre- and post-condition of CPMM

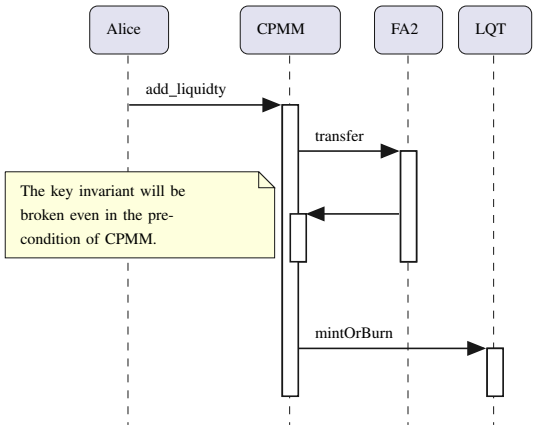

Fig. 6. Misbehaved FA contract

`total_supply` of LQT tokens with the sum of the `tokens` map, which represents the LQT token ledger. Although the LQT contract itself maintains this invariant, its relevance to CPMM's verification is paramount. Specifically, the correctness of `mintOrBurn` implementation, and hence the key invariant, depends on this relationship. The implementation updates `total_supply` to the absolute value of the sum of `total_supply` and `quantity` (the latter being negative when tokens are burned) to ensure the new value is a natural number. Thus, the new value's correctness is contingent upon the sum being non-negative. Given this invariant, we can infer that `total_supply + quantity` is non-negative, as `total_supply` equals `sum_of tokens`, which is greater than or equal to `tokens[target]`, and in turn greater than or equal to the negative `quantity`. A potential improvement to the implementation could include an explicit check for $\texttt{total\_supply} + \texttt{quantity} \geq 0$, which would allow for the omission of the third equation from the pre- and post-conditions, subsequently simplifying the verification process.

The history invariant requires additional investigation. Direct utilization of the key invariant is not feasible for CPMM due to the `add_liquidity` entrypoint, where a `transfer` call precedes a `mintOrBurn` call. This scenario, illustrated in Fig. 6, considers the FA2 as an unknown contract, thus necessitating an examination of cases where the `transfer` call results in a callback to the CPMM contract. This flow temporarily breaks the key invariant—`lqt_total` is updated, yet `total_supply` remains unchanged as the `mintOrBurn` call is pending. Therefore, it is crucial to remark that the key invariant does not perpetually hold during CPMM invocations.

```
predicate post (st : step) (gp : gparam)
                (c : ctx) (c' : ctx) =
  c'.cpmm_storage.Cpmm.lqt_address = Lqt.addr ∧
  c'.lqt_storage.Lqt.admin = Cpmm.addr ∧
  c'.lqt_storage.Lqt.total_supply
    = sum_of c'.lqt_storage.Lqt.tokens ∧
  let lqt_total
       = c.cpmm_storage.Cpmm.lqt_total in
  let total_supply
       = c.lqt_storage.Lqt.total_supply in
  let lqt_total'
       = c'.cpmm_storage.Cpmm.lqt_total in
  let total_supply'
       = c'.lqt_storage.Lqt.total_supply in
  if st.sender = Cpmm.addr then
    match gp with
    | Gp'Lqt'mintOrBurn quantity _ ->
        lqt_total' - total_supply' + quantity
         = lqt_total - total_supply
    | _ -> lqt_total' - total_supply'
           = lqt_total - total_supply
    end
  else lqt_total' - total_supply'
       = lqt_total - total_supply)
```

Fig. 7. Post-condition of LQT

While integrating FA2 into the set of known entities would be another design choice (since the standardized token contract used in deploying CPMM can be controlled), this approach is not pursued here. Our objective is to demonstrate that the key invariant can be derived independently of the FA contract's specifics. Directly applying the key invariant to the pre- and post-conditions of CPMM is feasible if FA2 is considered a known contract. However, this necessitates the inclusion of an FA contract description in the verification input, representing a trade-off decision.

b) LQT contract: The `pre` condition for the LQT contract is identical to that of the CPMM contract. Notably, `pre` does not impose any direct relationship between `lqt_total` and `total_supply`, thus circumventing the issue highlighted in Section V-B1. The condition `post`, as presented in Fig. 7, differs slightly from that of the CPMM. It specifies that the difference between `lqt_total` and `total_supply` aligns with the `quantity` of tokens minted or burned, but only when the `mintOrBurn` entrypoint is invoked from CPMM. This conditional adjustment ensures that the temporarily broken key invariant is recovered following LQT's execution by CPMM.

c) Unknown contracts: For unknown contracts, both pre- and post-conditions are the same as those defined for CPMM. This approach is based on the reasoning that each pre- and post-condition of known contracts (CPMM and LQT) forms a history invariant, assuming `st.sender` is an unknown contract. Consequently, we can verify that these conditions

hold for any unknown contracts, irrespective of their specific implementations, as elaborated in Section III.

VI. Theoretical Result

To clarify the validity of our verification approach, this section: (1) formalizes the inter-contract semantics of smart contracts in Tezos; (2) proposes a Hoare style proof system for verifying an inter-contract system; and (3) shows the correctness of the proof system. To focus on the essence, we make the following simplification in this section: (1) We include only operations for transferring tokens, (2) Each contract takes an integer as a parameter, and (3) Each contract keeps an integer storage value. Consequently, the syntax of operation lists, ranged over by $\vec{o}$, is defined as follows, where i, m, and a range over integers.

$$\vec{o} ::= [] \mid \texttt{Xfer}\,(i, m, a) :: \vec{o}$$

Typically, i is used for a general integer, m for the amount of XTZ (in a unit *mutez*), and a for the address of an account.

A. Operational semantics

We model the execution of operation lists as a relation between the blockchain states before and after an operation list is committed. A blockchain state is modeled by a pair of finite maps: a *ledger*, ranged by L, from addresses to mutezs, which represents the amount of XTZ owned by each account; and *stores*, ranged by S, from addresses to *scripts*, a pair of *code*, ranged over by c, and an integer, which represent contract code and a contract storage value, respectively. A resulting judgment is $\langle a \Rsh \vec{o} \mid L_1, S_1\rangle \Downarrow (L_2, S_2)$, which says the execution of $\vec{o}$ emitted by the sender a starting from the block state (L_1, S_1) successfully ends with the block state (L_2, S_2).

The derivation rules are obtained by a naive formalization of the execution model introduced in Section II. For instance, the following rule is for `Xfer`, where $f \lhd \{i \mapsto v\}$ denotes the finite map whose domain is identical to f except v at i, and the partial function $\mathfrak{I}$ models the Michelson virtual machine.

$$\frac{\begin{array}{c} m' \simeq L(a_1) \quad m' \geq m \geq 0 \quad L_1 = L \lhd \{a_1 \mapsto m' - m\} \\ m'' \simeq L_1(a_2) \quad L_2 = L_1 \lhd \{a_2 \mapsto m'' + m\} \\ (c, i_2) \simeq S(a_2) \quad (\vec{o}', i_3) \simeq \mathfrak{I}(c, a_1, a_2, m, i_1, i_2) \\ \langle a_2 \Rsh \vec{o}' \mid L_2, S \lhd \{a_2 \mapsto (c, i_3)\}\rangle \Downarrow (L', S') \\ \langle a_1 \Rsh \vec{o} \mid L', S'\rangle \Downarrow (L'', S'') \end{array}}{\langle a_1 \Rsh \texttt{Xfer}\,(i_1, m, a_2) :: \vec{o} \mid L, S\rangle \Downarrow (L'', S'')}$$

B. Proof system

We formalize our proof system following the style of the Hoare logic. The main judgment is $K \vdash a \Rsh \{P\}\vec{o}\{Q\}$, where P and Q are assertions written in a first-order logic by which we can mention a blockchain state via unique variables `ledger` and `stores`. In this judgment, the right side $\{P\}\vec{o}\{Q\}$ of $\Rsh$ is the standard Hoare triple: if the execution of $\vec{o}$ starting from a block state satisfying P, called *pre-condition*, successfully finishes, the modified block state at the end satisfies Q, called *post-condition*.

The behavior of $\vec{o}$ depends on the contracts deployed on a blockchain at the execution time of $\vec{o}$ and the account that issues the $\vec{o}$. To handle the dependency, we include a *block specification* K and a in the judgment. Among them, a denotes the address of the account that issues the operations. K gives an abstraction of *all* the contracts, including one appears in the future, on a blockchain, which is a total map from addresses to triples consisting of parameterized assertions: *code approximation* $\mathcal{C}$, *invocation pre-condition* $\mathcal{P}$, and *invocation post-condition* $\mathcal{Q}$. A code approximation $\mathcal{C}$ describes a functional property of the contract at an address. $\mathcal{P}$ and $\mathcal{Q}$ represent blockchain states before and after the contract is invoked at an address. So, $\mathcal{C}$ corresponds to `Spec` scope, and $\mathcal{P}$ and $\mathcal{Q}$ correspond to `pre` and `post` predicates of an iCon input file. A block specification comes with a twist so that $\mathcal{P}$ and $\mathcal{Q}$ are consistent with $\mathcal{C}$ rather than an actual code. This is the key abstraction to uniformly handle known contracts and infinitely existing unknown contracts; that is, we give unknown contracts the code approximation logical truth $\top$, which trivially covers the behavior of arbitrary code. So, not knowing the future, we can still give meaningful K.

We do not show the proof rules for the main judgment. Being able to know every contract pre- and post-condition from K, we can naively construct the proof rules since $\vec{o}$ is simply a sequence of contract calls. Notice that the proof rules for the main judgment only use $\mathcal{P}$ and $\mathcal{Q}$ in K.

The characteristic part of our proof system is the consistency checking of a block specification, given by two judgments $K \vdash a$ and $\vdash K$. The latter judgment holds iff $K \vdash a$ for any a, that is straightforward. So, the former is fundamental. The judgment $K \vdash a$ means that the behavior of the account at a meets the abstraction described in K, formally derived by the following unique rule.

$$\frac{x_1\, x_2\, x_3\, x_4\, x_5\, x_6 \,\#\, FV(K) \qquad (\mathcal{C}, \mathcal{P}, \mathcal{Q}) = K(a) \qquad \left(\begin{array}{l} K \vdash a \Rsh \{\mathcal{P}()[x_1/\texttt{ledger}, x_2/\texttt{stores}] \wedge \\ \mathcal{C}(x_3, a, x_4, x_5, x_6, \vec{o}, \texttt{stores}[a]) \wedge x_1 = \\ \texttt{ledger}[a -= x_4] \wedge x_2 = \texttt{stores}[a \mapsto \\ x_6]\}\vec{o}\{\mathcal{Q}(x_1, x_2, x_3, a, x_4, x_5)\} \text{ for any } \vec{o} \end{array}\right)}{K \vdash a}$$

The first premise supposes the variables newly introduced are distinguished from the existing ones, and the second premise takes the triple abstracting the contract deployed at a whose consistency is being checked. Those two premises are always satisfiable, so the third long premise, enclosed by parentheses, is the main condition to be checked. The third premise checks the behavior of internal operations, namely $\vec{o}$, that the contract issues when receiving x_4 amounts of XTZ with the bundled parameter x_5 from the account at x_3. The pair of auxiliary variables x_1 and x_2 represents the blockchain state at Line 1 in Algorithm 1, and x_6 represents the original storage value of the contract. We summarize the assertions: the pre-condition specifies the condition expected at Line 8, and the post-condition specifies the one at Line 26. To check all possibilities of the issue, we quantify the premise over any $\vec{o}$ and constrain it by the code approximation $\mathcal{C}$ in the pre-

condition.

We remark that we cannot construct a naive proof tree by using the proof rules because the third premise of the proof rule for $K \vdash a$ demands the main judgment holds for any $\vec{o}$, and we need to check $K \vdash a$ for any a. However, we can still check the consistency using meta-level proof. We give all unknown contracts the code approximation $\top$ and a unique pair of $\mathcal{P}$ and $\mathcal{Q}$, which results in a block specification K with a finite domain since we suppose that the number of known contracts is finite. Now, we check the consistency of each triple in the domain, as we have summarized in Section III. iCon realizes this meta-level proof formally in Why3, and therefore, the correctness of the verification is supported by the soundness property of the proof system.

C. Soundness

Here, we show the soundness statement.

Theorem 1 (Soundness)**.** *Suppose* $\vdash K$. *If* $K \vdash a \Rsh \{P\}\vec{o}\{Q\}$, *then* $K \models a \Rsh \{P\}\vec{o}\{Q\}$.

The soundness statement is shown above, which looks like the usual one, except we need an explanation for the validity of partial correctness assertion $K \models a \Rsh \{P\}\vec{o}\{Q\}$. Roughly speaking, the judgment says, for any (L, S) and (L', S') such that $\langle a \Rsh \vec{o} \mid L, S\rangle \Downarrow (L', S')$, if blockchain state (L, S) satisfies the pre-condition P and the block specification K, then (L', S') satisfies the post-condition Q. One unusual part will be S needs to satisfy K; that is, every deployed contract's code needs to satisfy the corresponding code approximation.

VII. Related work

A. Smart contract verification

After the infamous DAO attack, which is considered the most significant incident in blockchain history, many verification techniques have been proposed [6]–[12], [14], [22]–[28]. Among them, we focus on 2Vyper [9], Solicitous [10], and ConCert framework [22], [23], which are much related to our work.

2Vyper [9] targets the Vyper language, a smart contract language for the Ethereum platform, and supports inter-contract properties. It also provides resource-based assertions, by which users can quickly write various assertions common to cryptoassets. Aside from the assertion language, 2Vyper uses Viper [29], an SMT-based program verification platform, as its backend. Therefore, their verification framework is quite similar to ours. Their verification approach is similar to ours, but they use three kinds of primitive specifications: *transitive segment constraints*, *functional constraints*, and *functional properties*. Roughly speaking, our contract pre- and post-condition subsume both functional constraints and functional properties. So, we are curious if the transitive segment constraints are essential, as we can adapt some case study applications from their work. Because of the difference between the execution models of Ethereum and Tezos smart contracts, we still need more investigation into the question.

Solicitous [10] is an automated assertion checker for Solidity. Its novelty lies in that a programmer simply writes Solidity code, which will involve assertions as usual, and then Solicitous can check if the assertions always hold without additional specifications, like contract invariants. Solicitous logically encodes a given program's behavior into constrained Horn clauses (CHC) [30] form and lets an external solver solve them, during which sufficient invariants are inferred. They have shown the ability of Solicitous by a comprehensive experiment; that is, showing how many contracts deployed on the Ethereum chain can be verified by Solicitous. The result is that roughly 30% of the contracts have been verified. Although it is, in fact, an outstanding result, we are also interested in how Solicitous well works for individual contracts, especially having a complex invariant.

ConCert [22], [23] is a Coq framework for smart contract verification. In [15], the authors formalize Dexter2 and verify the same property we have shown in Section V using the framework. That is why we chose Dexter2 as the main case study. A noticeable difference from the result of our case study is that they need to write thousands of lines of proof code to verify the property. It is a heavy burden, though we can have full control over a verification, meaning that we never be bothered by the whimsical behavior of an external theorem prover. The proof strategy they use is also different from our verification approach. They firstly show an invariant between a single contract state and queued incoming and outgoing messages (operations) for each known contract and then derive a multi-contract property by relating the invariants via the message queue. So, even though they also use invariants and the final result they got is the same one, the invariants are very different from ours.

VIII. Conclusion

We have presented a novel approach to verifying inter-transaction properties in smart contracts on the Tezos blockchain platform. Our work introduces a program logic designed to handle these properties even in the presence of unknown contracts. The verification method is based on this program logic, employing an abstraction technique for unknown contracts. We have also implemented a verifier iCon on the Why3 verification framework and demonstrated its effectiveness through various case studies, such as the decentralized exchange service Dexter.

A potential direction for future research is integrating our method with existing verifiers for single contracts [11]–[14]. Currently, iCon requires users to provide a summary of a contract's behavior, represented as $\mathcal{C}$, and an upper bound of the number of internal operations issued by the contract, denoted as N_i. We anticipate that this information could be automatically synthesized from the verification results of each contract when using single-contract verifiers.

Another direction is to support origination operations, which are less used but have non-trivial behavior, namely creating a new smart contract account.

References

[1] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, Sep. 1997. [Online]. Available: https://firstmonday.org/ojs/index.php/fm/article/view/548

[2] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[3] V. Buterin. (2014) Ethereum: A next-generation smart contract and decentralized application platform. [Online]. Available: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf

[4] G. Wood. (2022) Ethereum: A secure decentralised generalised transaction ledger berlin version beacfbd. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[5] L. Goodman. (2014) Tezos — a self-amending crypto-ledger. [Online]. Available: https://tezos.com/whitepaper.pdf

[6] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, vol. 10, pp. 57 037–57 062, 2022.

[7] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1661–1677.

[8] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum," in *Financial Cryptography and Data Security*, I. Goldberg and T. Moore, Eds. Cham: Springer International Publishing, 2019, pp. 446–465.

[9] C. Bräm, M. Eilers, P. Müller, R. Sierra, and A. J. Summers, "Rich specifications for ethereum smart contract verification," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3485523

[10] R. Otoni, M. Marescotti, L. Alt, P. Eugster, A. Hyvärinen, and N. Sharygina, "A solicitous approach to smart contract verification," *ACM Trans. Priv. Secur.*, vol. 26, no. 2, mar 2023. [Online]. Available: https://doi.org/10.1145/3564699

[11] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson, "Mi-Cho-Coq, a framework for certifying Tezos smart contracts," in *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part I*, ser. Lecture Notes in Computer Science, vol. 12232. Springer, 2019, pp. 368–379.

[12] Y. Nishida, H. Saito, R. Chen, A. Kawata, J. Furuse, K. Suenaga, and A. Igarashi, "Helmholtz: A verifier for tezos smart contracts based on refinement types," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. F. Groote and K. G. Larsen, Eds. Cham: Springer International Publishing, 2021, pp. 262–280.

[13] The Archetype development team, "Archetype," Completium, 2023. [Online]. Available: https://archetype-lang.org/

[14] L. P. A. da Horta, J. S. Reis, S. M. de Sousa, and M. Pereira, "A tool for proving michelson smart contracts in why3*," in *IEEE International Conference on Blockchain, Blockchain 2020, Rhodes, Greece, November 2-6, 2020*. IEEE, 2020, pp. 409–414. [Online]. Available: https://doi.org/10.1109/Blockchain50366.2020.00059

[15] E. H. Nielsen, D. Annenkov, and B. Spitters, "Formalising decentralised exchanges in coq," in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 290–302. [Online]. Available: https://doi.org/10.1145/3573105.3575685

[16] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Programming Languages and Systems*, M. Felleisen and P. Gardner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128.

[17] Nomadic Labs. (2021) Dexter flaw discovered; funds are safe. [Online]. Available: https://research-development.nomadic-labs.com/dexter-flaw-discovered-funds-are-safe.html

[18] ——. (2021) A technical description of the dexter flaw. [Online]. Available: https://research-development.nomadic-labs.com/a-technical-description-of-the-dexter-flaw.html

[19] B. Meyer, *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. [Online]. Available: http://www.eiffel.com/doc/oosc/page.html

[20] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, p. 1811–1841, nov 1994. [Online]. Available: https://doi.org/10.1145/197320.197383

[21] K. R. M. Leino and W. Schulte, "Using history invariants to verify observers," in *Programming Languages and Systems*, R. De Nicola, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 80–94.

[22] D. Annenkov, J. B. Nielsen, and B. Spitters, "Concert: A smart contract certification framework in coq," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 215–228. [Online]. Available: https://doi.org/10.1145/3372885.3373829

[23] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters, "Extracting smart contracts tested and verified in coq," in *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 105–121. [Online]. Available: https://doi.org/10.1145/3437992.3439934

[24] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *Financial Cryptography and Data Security*, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds. Cham: Springer International Publishing, 2017, pp. 520–535.

[25] I. Grishchenko, M. Maffei, and C. Schneidewind, *A Semantic Framework for the Security Analysis of Ethereum Smart Contracts*, 04 2018, pp. 243–269.

[26] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 204–217.

[27] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 66–77. [Online]. Available: https://doi.org/10.1145/3167084

[28] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments*, S. Chakraborty and J. A. Navas, Eds. Cham: Springer International Publishing, 2020, pp. 161–179.

[29] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62.

[30] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, *Horn Clause Solvers for Program Verification*. Cham: Springer International Publishing, 2015, pp. 24–51. [Online]. Available: https://doi.org/10.1007/978-3-319-23534-9_2