U.PORTO

FACULDADE DE CIÊNCIAS - Ano Letivo 2020/2021

DEPARTAMENTO DE CIÊNCIA DE COMPUTADORES

Projeto

Property-based testing of ERC-721 Ethereum smart contracts

Isac Daniel de Figueiredo Novo
200403278

Índice

1 Introduction **2**

1.1 Motivation 2

1.2 Problem Statement 2

2 Background **3**

2.1 Blockchain 3

2.2 Ethereum 4

2.3 Smart Contracts 5

2.4 ERC-721 Non-Fungible Token Standard 6

2.4.1 Events 7

2.4.2 Functions 8

3 Test Framework **10**

3.1 Property-Based Testing 10

3.2 Property-Based Testing with Brownie 11

3.2.1 The Hypothesis framework 11

3.2.2 Test Execution 12

3.3 Implementation 13

3.3.1 State Machine 14

3.3.2 Class Hierarchy and Project Organization 17

4 Evaluation **18**

4.1 Methodology 18

4.2 Overall Results 18

4.3 Bug Analysis 20

5 Conclusion **22**

1 Introduction

1.1 Motivation

Blockchain technology has been the subject of increasing interest by researchers and industry entrepreneurs alike. This is due in big part to BitCoin's major success as a cryptocurrency, but also thanks to the distributed protocol inherent to the blockchain technology. Yet, despite cryptocurrencies still being the leading topic of discussion regarding blockchains, this emergent technology offers a wide array of possible applications.

While some blockchain applications can still be financial in nature, its distributed public ledger scheme makes it ideal for any application that relies on the bookkeeping and managing of assets. These can be, but are not limited to, managing physical or digital property, domain names, tracking votes, and so on.

These general-purpose applications of blockchains are enabled with the deployment of what is know as smart contracts. Literally, a smart contract is program code that implements a set of standard rules for transactions and a plethora of arbitrarily user-defined functions.

Ethereum [23] is the leading blockchain for the development and deployment of smart contracts, that are written in languages such as Solidity [19] and Vyper [22], whose target bytecode is the Ethereum Virtual Machine (EVM). The EVM stores both a smart contract's bytecode and state, executing the contract's functions as blockchain transactions, that usually involve manipulating Ether - Ethereum's own cryptocurrency.

As promising as they are, being decentralized applications, smart contracts are still in the early stages of its constant development, and with an ever-growing macrocosm of valuable assets, are prone to a number of errors and malicious attacks. Aside from these, the implementation of Ethereum contracts is oftentimes unreliable. For instance, albeit the most widely used language for writing contracts, Solidity is still on its alpha stage of development. Vulnerabilities also seem to arise from the disparity between Solidity's semantics and blockchain specifics. Concretely, the lack of constructs that handle domain-specific aspects, such as the persistence of smart contracts in the blockchain, their interaction amongst themselves and with external services, or the fact that computation steps are registered on a public data structure - at which point they can be unpredictably reordered or delayed.

1.2 Problem Statement

There are several documentations for Ethereum that aims to improve smart contract functionality and implementation; one such document - ERC-721 - provides a standard interface for non-fungible tokens.

In this project, we tested five Ethereum smart contracts for errors. The contracts in question were 0xcert, Decentraland, Ethereum Name Service, OpenZeppelin, and Su Squares; that implement the ERC-721 standard.

Verification on those contracts was done by property-based testing, an approach that generates a

random number of test cases according to predefined strategies and derives test cases following a set of given rules following a model of correctness which, as mentioned, was based on the ERC-721 standard. By employing this method, a programmer can focus their attention on the desired properties of the software alongside a model for input generation, rather than having to construct a finite set of test cases - as is the case for unit testing.

This approach allows for more tests to be executed with less lines and code, and does not have to rely on the programmers insight regarding the code. Thus, being more efficient at deriving tests that detect edge cases and less common interaction that could otherwise be missed.

The entire project code and environment - including the five tested contracts - is available as a public GitHub repository [9].

The remainder of this report is organised as follows. Section 2 provides the necessary background for this project, regarding blockchains, smart contracts, and the ERC-721 standard. Section 3 describes the ERC-721 test framework implemented in this project. Section 4 describes an evaluation of the test framework for 5 ERC-721 implementations. Finally, Section 5 concludes the report with an overall assessment.

2 Background

2.1 Blockchain

A blockchain is a data structure akin to a distributed public ledger in which all committed transactions are stored in a continuously growing chain of blocks, hence the name. BitCoin [bitcoin], proposed in 2008 and later implemented in 2009 by Satoshi Nakamotom, was the first decentralized cryptocurrency employing a peer-to-peer network as solution for the double-spending problem [14] - that is, the potential flaw in a digital currency system by which the same single digital token can be spent more than once [4].

Bitcoin was also the first digital asset without any backing or intrinsic value as well as simultaneously no centralized issuer or controlling authority. By working with a collection of nodes, peer-to-peer, without the need for a trusted third-party authority, blockchains offer decentralization, persistency, anonymity, and still auditability.

A blockchain, as the name suggests, consists of a series of appending blocks. As illustrated by Figure 1, each block holds an hash value of the previous one that serves both as pointer to it - the parent block - thus linking them in a chain, as well as to prevent the tampering of a single block without compromising the entire chain down the line. A Unix time timestamp of the block's creation that, while also being used as a source for the block hash, is to be validated by other clients and difficult adversarial manipulation, allowing for a certain clock skew. A nonce, a central part to the Proof of Work consensus done by the network, has to be determined so that the block's hash value is below a certain target value. And finally, as a complete list of all transaction records performed up to that point in the form of a Merkle root hash.

By using a Merkle tree, where every subsequent block confirms its parent via the cryptographic hash,

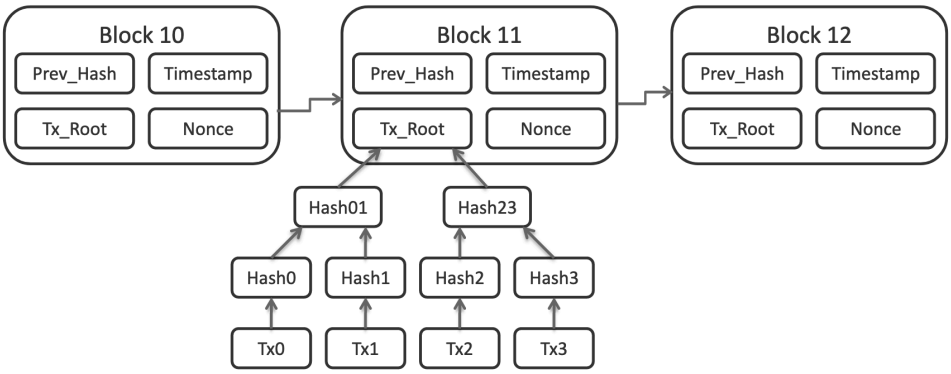


Figure 1: Simplified depiction of a blockchain's block data fields as well as the Merkle tree's hash

all the way up to the first block on the chain - the genesis block - blockchains offer robustness of design and resistance to data corruption. This is thanks to the aforementioned block structure and the cryptographic security therein, as well as the underlined consensus protocols based on Proof of Work mechanisms that offer high byzantine fault tolerance - that is to say, are not very susceptible to wrong information or errors exhibited by some of the nodes on the distributed system.

2.2 Ethereum

Other than for cryptocurrencies, blockchains can be employed on many different applications controlled by code implementing arbitrary rules - smart contracts. Of particular interest to this project are smart contracts that govern non-fungible assets that are tied to physical or digital property.

Ethereum [3, 23] is a decentralized, open-source blockchain that supports smart contracts. It is currently the second largest cryptocurrency by market capitalization and the most used blockchain after Bitcoin. A programmable blockchain, it is a community-built technology behind not only the development of thousands of blockchain applications, as well as of the built-in Ether (ETH) tradeable cryptocurrency. The latter being used to pay for the fees associated with running code by the former.

The blockchain aims to provide a protocol for building decentralized applications, or DApps, with different sets of useful trade-offs and an emphasis on the swift development of diverse yet secure applications able to interact amongst themselves. Without the need for a central administrative authority, it falls to a peer-based network to enforce good practices, as stipulated by guidelines in akin to Request for Comments publications, as will be delved into in Section 2.4.

In order to accomplish this, the blockchain model offers the built-in Turing-complete Ethereum Virtual Machine, with core support for programming languages such as Vyper [22] and Solidity [19] - the one used on this project. This allows for the development of smart contracts encoding arbitrary state transition functions. Each application having its own unique set of rules for ownership, transaction formats, and state transaction functions.

2.3 Smart Contracts

The Ethereum blockchain's state is comprised of account objects, each defined by a 20-byte address. State transitions occur with the direct transfer of value and information between account, as is depicted on Figure 2.

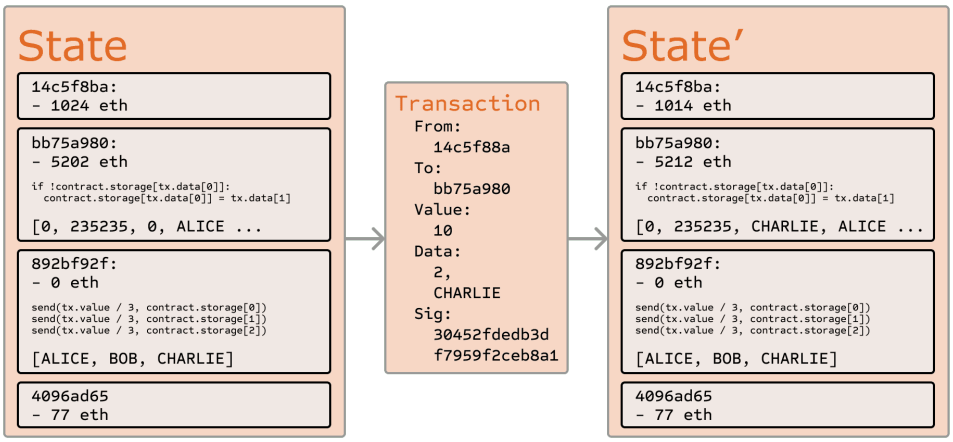


Figure 2: Ethereum state transition [3]

Ethereum accounts can be either be externally owned, controlled by private keys and with no associated code, or smart contract accounts, regulated by EVM bytecode that is associated to the account. The state of an account is defined by four fields: a nonce counter to assure that each transaction is only processed once, the account's current ether balance, the account's code in the case of smart contract accounts, and the account's storage (initially empty).

The code of a smart contract is activated every time the account receives a message, enabling it to read and write to internal storage and send follow-up messages or create subsequent contracts. All messages sent and received as part of code execution have several fields. Standard ones for cryptocurrency include recipient, sender's signature, and the amount to be transferred. There is also an optional data field that is meant to be accessed by the receiver if needed, depending on the application. For example, for a domain name service, a name and IP address could be sent in order to perform a registration between the two.

Finally, two last fields, known as the gas limit and the gas price, set the maximum number of computational steps that the transaction execution is allowed to perform and the fee to be paid by the sender per computational steps. All transaction fees, associated with a code execution gas cost, are paid with Ether. The gas mechanism inhibits denial-of-service, by preventing accidental or hostile heavy computation (e.g. infinite loops). Computational steps in a transaction are billed according to the EVM bytecode they execute, where each EVM instruction has its own gas price.

Figure 3: In CryptoKitties, an Ethereum-based DApp that uses the ERC-721 standard, each NFT represents a unique digital asset - A CryptoKitty [6]

2.4 ERC-721 Non-Fungible Token Standard

ERC stands for Ethereum Request for Comments and are technical standards for Ethereum-based tokens, available online as EIPs - Ethereum Improvement Proposals [10]. EIPs include what are known as core protocol specifications, that encompass those that have been implemented and released, or those that are planed to be so, alongside client APIs and contract standards.

ERCs comprise application-level standards and conventions for Ethereum including, but not restricted to, smart contract and token standards. And, while ERC-20 specifies the standard token interface, providing a model implementation for a smart contract token API, ERC-721 does so for non-fungible tokens [8]. There are two major standards for tokens: ERC-20 [21] for fungible tokens (FTs), and ERC-721 [8] for non-fungible tokens (NFTs). FTs represent different quantities of identical (fungible) assets, and are many times employed for the implementation of crypto-currencies on top of Ethereum. NFTs in turn can represent ownership of several kinds of distinct assets, for instance physical property such as houses or unique artwork, or collectibles like virtual pets or game cards. Figure 3 provides an example of a real-world usage for NFTs.

The ERC-721 standard provides basic functionality for the tracking and transferring of NFTs in its smart contract API, taking into account not only when the tokens are transacted by their individual owners, as well as by consigned third parties. These authorized mediators can be brokers, wallets, or auctioneers, and are known as operators in this context. It also allows for these broker/wallet/auction applications to work with any NFT on Ethereum, while providing for both simple smart contracts as well as those that track a large number of NFTs.

Let us illustrate how ERC-721 contracts work using the code of a well-known reference implementation, OpenZeppelin [17]. Listing 1 contains the code of the Solidity interface for ERC-721 tokens, and Listing 2 contains a fragment of corresponding implementation.

As shown in Listing 1, a contract's interface is expressed by events and functions. Events correspond to information emitted onto the blockchain transaction logs by the code of a smart contract. Event allow for efficient queries and provide lower-cost data storage when the data is not required to be accessed on-chain. Functions in turn correspond to code that can be called within a transaction. We next describe the events and functions that are defined by the ERC-721 standard.

```
pragma solidity ^0.5.2;

import "../../introspection/IERC165.sol";

/**
 * @title ERC721 Non-Fungible Token Standard basic interface
 * @dev see https://eips.ethereum.org/EIPS/eip-721
 */
contract IERC721 is IERC165 {
    event Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId);
    event Approval(address indexed _owner, address indexed _approved, uint256 indexed _tokenId
    );
    event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved);

    function balanceOf(address owner) public view returns (uint256 balance);
    function ownerOf(uint256 tokenId) public view returns (address owner);

    function approve(address to, uint256 tokenId) public;
    function getApproved(uint256 tokenId) public view returns (address operator);

    function setApprovalForAll(address operator, bool _approved) public;
    function isApprovedForAll(address owner, address operator) public view returns (bool);

    function transferFrom(address from, address to, uint256 tokenId) public;
    function safeTransferFrom(address from, address to, uint256 tokenId) public;

    function safeTransferFrom(address from, address to, uint256 tokenId, bytes memory data)
    public;
}
```

Listing 1: OpenZeppelin's IERC-721 (interface) standard

2.4.1 Events

ERC-721 defines the following events:

— Transfer(**address indexed** _from, **address indexed** _to, **uint256 indexed** _tokenId)

- Emitted whenever ownership of any NFT is changed by any mechanism, including during the creation or destruction of a token - by resorting to address 0 for sender or receiver, respectively. An exception to this rule is during contract creation, when any number of NFTs can be created and assigned without emitting Transfer.
- Whenever Transfer is fired, all approved addresses for that token are reset.

— Approval(**address indexed** _owner, **address indexed** _approved, **uint256 indexed** _tokenId)

- Emitted whenever an approved address for a given NFT is changed or reaffirmed. The zero address indicates that there are no approved addresses for that token. It is also emitted alongside Transfer to reset the approved address.

— ApprovalForAll(**address indexed** _owner, **address indexed** _operator, **bool** _approved)

- Emitted when an operator is approved for all NFTs of the holder.

```
contract ERC721 is ERC165, IERC721 {
    mapping (uint256 => address) private _tokenOwner;
    mapping (uint256 => address) private _tokenApprovals;
    mapping (address => Counters.Counter) private _ownedTokensCount;
    mapping (address => mapping (address => bool)) private _operatorApprovals;
    ...
    function balanceOf(address owner) public view returns (uint256) {
        require(owner != address(0));
        return _ownedTokensCount[owner].current();
    }
    function ownerOf(uint256 tokenId) public view returns (address) {
        address owner = _tokenOwner[tokenId];
        require(owner != address(0));
        return owner;
    }
    function approve(address to, uint256 tokenId) public {
        address owner = ownerOf(tokenId);
        require(to != owner);
        require(msg.sender == owner || isApprovedForAll(owner, msg.sender));
        _tokenApprovals[tokenId] = to;
        emit Approval(owner, to, tokenId);
    }
    function getApproved(uint256 tokenId) public view returns (address) {
        require(_exists(tokenId));
        return _tokenApprovals[tokenId];
    }
    function setApprovalForAll(address to, bool approved) public {
        require(to != msg.sender);
        _operatorApprovals[msg.sender][to] = approved;
        emit ApprovalForAll(msg.sender, to, approved);
    }
    function isApprovedForAll(address owner, address operator) public view returns (bool) {
    return _operatorApprovals[owner][operator]; }
    function transferFrom(address from, address to, uint256 tokenId) public {
        require(_isApprovedOrOwner(msg.sender, tokenId));
        _transferFrom(from, to, tokenId);
    }
    function safeTransferFrom(address from, address to, uint256 tokenId, bytes memory _data)
    public {
        transferFrom(from, to, tokenId);
        require(_checkOnERC721Received(from, to, tokenId, _data));
    }
    function _transferFrom(address from, address to, uint256 tokenId) internal {
        require(ownerOf(tokenId) == from);
        require(to != address(0));
        _clearApproval(tokenId);
        _ownedTokensCount[from].decrement();
        _ownedTokensCount[to].increment();
        _tokenOwner[tokenId] = to;
        emit Transfer(from, to, tokenId);
    }
    ...
}
```

Listing 2: ERC721 OpenZeppelin implementation overview

2.4.2 Functions

— balanceOf(**address** _owner) **external view returns** (**uint256**)

- Returns the number of NFTs owned by the address, possibly zero. Throws exceptions for 0 address tokens, since those are considered invalid (Listing 2, lines 7-10).

— ownerOf(**uint256** _tokenId) **external view returns** (**address**)

- Returns the address corresponding to the given NFT's owner. Zero address tokens are, once again, considered not valid (2, lines 11-15).

— transferFrom(**address** _from, **address** _to, **uint256** _tokenId) **external payable**

- Unsafe version of safeTransferFrom. Transfers ownership of an NFT. Sender must be token's current owner, an authorized operator, or the approved address for the token that is to be transferred. Furthermore, the owner address must be that of the token's holder. Also, the token needs to be a valid one and remain that way, therefore the receiver cannot be the zero address (Listing 2, lines 33-36 and 41-49).

- The function is unsafe in the sense that it is up to the caller to check that the receiver is able to receive NFTs. If not, tokens may be permanently "lost" by becoming associated to accounts that prevent further manipulation of the token.

—safeTransferFrom(**address** _from, **address** _to, **uint256** _tokenId) **external payable**
and safeTransferFrom(**address** _from, **address** _to, **uint256** _tokenId, **bytes** data) **external payable**

- Behaves like transferFrom, but disallows "unsafe" transfers by checking if the receiver is a smart contract able to receive ERC-721 tokens, throwing an exception if not (Listing 2, lines 37-40). For that purpose, a special function called onERC721Received is called for the receiver's address

- The second variant of the method (not shown in the listing) has an additional data parameter that may be passed on top the onERC721Received call.

— approve(**address** _approved, **uint256** _tokenId) **external payable**16

- Changes or reaffirms the approved address for an NFT. The zero address is used to indicate that there is no approved address for a given token. The sender must be the owner or authorized operator, otherwise an exception is thrown (Listing 2, lines 16-22).

— setApprovalForAll(**address** _operator, **bool** _approved) **external**

- Authorizes or revokes the authorization of a third party operator for all of the sender's tokens. A contract must allow multiple operators per holder (Listing 2, lines 27-31).

— getApproved(**uint256** _tokenId) **external view returns** (**address**)

- Returns the approved address for a single NFT, zero if there is none. The token must have a valid id, otherwise an exception is thrown (Listing 2, lines 23-26).

— isApprovedForAll(**address** _owner, **address** _operator) **external view returns** (**bool**)

- Queries if a given address is an approved operator for the given NFT (Listing 2, lines 32-32).

3 Test Framework

3.1 Property-Based Testing

Property-based testing (PBT) is a methodology for software testing that automatically generates test cases using random inputs according to a model of correctness for the software under test. PBT looks for examples of incorrect behavior model, and then, in a process known as shrinking, tries to reduce the examples' complexity (e.g., in terms of values and example length) to facilitate human understanding of the bugs that may be staked. The PBT approach was born with QuickCheck for Haskell [5], and rapidly become a popular approach for other languages, e.g., Hypothesis for Python [13], or Scalacheck for Scala and Java [15] among many others.

PBT contrasts with unit example-based testing, also known as unit testing, in which a pre-generated finite set of inputs are respectively fed to a program and the expected outputs matched against the execution results. In comparison, PBT may generate a vast quantity of test cases in automated manner, which may make it easier to reproduce edge-case scenarios, increase code coverage, and cover interactions that may easily be missed by a programmer when coding unit tests.

Using PBT, a programmer may focus on specifying the properties of interest and input generation constraints through a model, and an arbitrary number test cases may be generated at random in model-driven manner. Shrinking is the other important aspect of PBT. Shrinking tries to automatically simplify test cases, when a violation of the PBT model is found. The purpose is to narrow the test cases in an attempt to locate the boundary values and/or interactions by which a property fails. This also helps with producing human readable examples for such failing examples.

Two distinct types of PBT are normally considered: stateless and stateful PBT. Stateless PBT verifies single interactions with the software under test, while stateful PBT considers sequences of interactions while keeping track of a "model state" of the software being tested. Stateful PBT is normally expressed by rule-based state machines, such that each rule implements an interaction of interest and test cases are formed by a sequence of several rules with varying length.

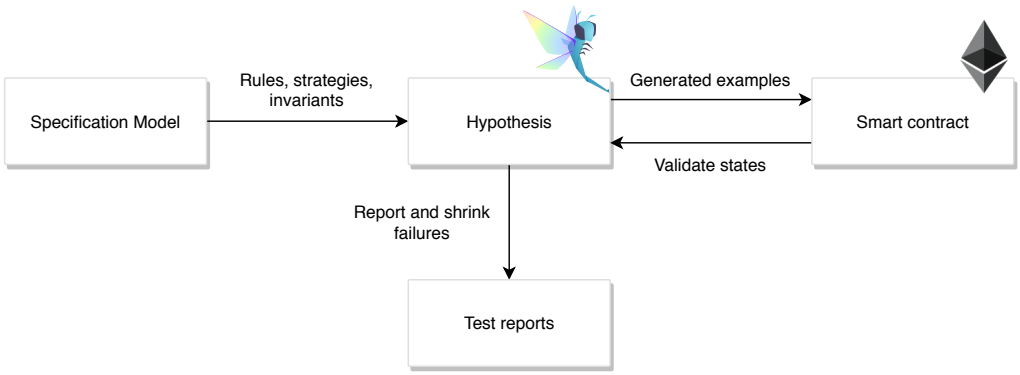


Figure 4: Schematic of the employed PBT methodology.

3.2 Property-Based Testing with Brownie

In this project, we employed the PBT approach for verifying ERC-721 contracts, following up on previous work that used PBT with ERC-20 contracts [18]. We make use of Brownie [2], an Ethereum Virtual Machine smart contract development and testing framework. Brownie employs the Hypothesis Python library for PBT [13] and the Ganache development blockchain [12].

A synopsis of this approach is illustrated in Figure 4 and described bellow:

- Brownie resorts to the Hypothesis framework for generating examples and validating state transition on the smart contract blockchain to be tested [2, 13].
- It then reports falsifiable test case instances or, in other words, generated inputs for which the ascertained properties of a functions did not match an expected or desired behaviour
- Alternatively, it can conduct shrinking to determine the smallest established failing case, reducing report complexity and promoting human readability, as previously mentioned.

Stateful test execution by Hypothesis, relies on a state machine class which defines an initial state, a number of structure-outlining actions for execution, and optional intransgressible invariants. These are all defined by rules, which are class methods that draw values from strategies and passes them to user defined functions, some examples of which are detailed further below. These rules can then be chained together and interact with each other in all sorts of ways.

3.2.1 The Hypothesis framework

The major considerations of this methodology, specifically in relation with the Hypothesis framework, are as follows:

- **Strategies**
  - We specified strategies in Brownie for the generation of random test cases, be it unsigned 256 bits integers for token addresses, or booleans for `setApprovalForAll`. Then, the generated examples are fed to the smart contract blockchain and consequently validated thanks to the Hypothesis framework.
- **Rules**
  - A desirable trait of this PBT library, is that other than having to test for properties such as the associative, commutative, or distributive properties, we can perform data-driven and interaction-driven testing as well. Specifically and respectively, stateless and stateful testing, where it is possible to ascertain the state transitions sustained by the EVM blockchain as a result of the transaction induced by the smart contract.
  - While for stateless testing a model for a single expected state transition is applied, for stateful-driven testing a sequence of defined actions are ran in a number of different ways in an attempt to reproduce a failure. Stateful testing is therefore suitable for complex contracts with many possible interactions.

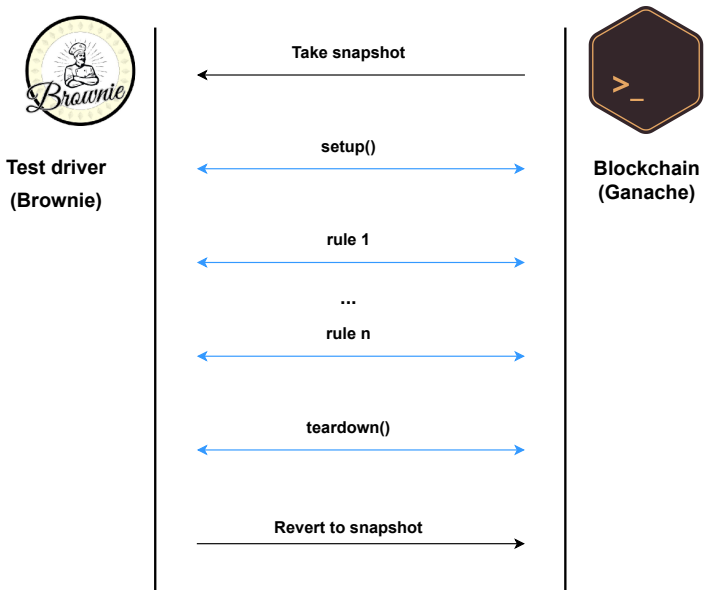


Figure 5: Brownie stateful testing procedure diagram

- **Coverage**
  - Lastly, the produced test reports, in lieu of identifying falsifiable examples, where the state transition did not occur as supposed, can produce a coverage report. This report denotes the perceptual portion of the contract code that was executed during testing, as well as highlights these relevant code lines.
  - Coverage analysis is useful in designing future test models - be it by adapting or even creating new rules, strategies, and so forth - that incorporate more of the smart contract to be tested.

3.2.2 Test Execution

The steps regarding this procedure are summarized in Figure 5. As illustrated, stateful text execution follows a straight sequence, albeit allowing room for variance.

Each test cycle begins with a snapshot of the blockchain, followed by running a setup phase and then a random and variable (user-defined) number of rules. The cycle terminates either once all the rules (the number of test execution steps that were set) are done, or immediately as soon as an error is found.

Regardless, the state machine is teared down and the blockchain is reverted to the taken snapshot.

As previously mentioned, rules can be executed with our without shrinking or, alternatively, a coverage report can be produced instead.

The stages of this procedure are detailed below, both in regards to each single test cycle, as well as the entire execution:

One-time initialization

- Prior to taking a snapshot of the blockchain, the `__init__` method, if present, passes external data into the state machine, that persists throughout every test for that execution series.

Text execution cycle (per Hypothesis internally generated example)

- At the beginning of each test run, immediately after blockchain reversion to the taken snapshot, the `setup` method is called. This method only affects the current test case.
- The test case example is formed by a randomly generated sequence of rule method invocations. A test failure may results from a failed assertion in one of the rules in the sequence, in which case the remaining rules are not executed. If no assertion fails over the entire sequence, the test succeeds.
- `teardown` is called after the rule sequence, regardless of whether the test case succeed or not, before chain reversion via the taken snapshot. This method can also be seen on Listing 4.
- These previous three steps are repeated for fixed number of times defined by the number of maximum examples. Note that the execution does not end on the first test failure, there can be multiple test failures.

3.3 Implementation

```
class StateMachine:
    # Input generation strategies
    . . .
    # Test lifecycle methods
    def __init__(self, wallets, contract, DEBUG=None):
        . . .
    def setup(self):
        . . .
    def teardown(self):
        . . .
    # Rules
    def rule_transferFrom(self, st_owner, st_receiver, st_token, st_sender):
        . . .
    def rule_safeTransferFrom(self, st_owner, st_receiver, st_token, st_sender):
        . . .
    def rule_approve(self, st_sender, st_token, st_receiver):
        . . .
    def rule_setApprovalForAll(self, st_sender, st_receiver, st_bool):
        . . .
    # Auxiliary Assertion methods
    def verifyOwner(self, tokenId):
       . . .
    def verifyBalance(self, wIdx):
       . . .
    def verifyApproved(self, token):
        . . .
    # other assertion methods . . .
```

Listing 3: ERC-721 state machine - methods overview.

3.3.1 State Machine

An overview of the state machine methods is shown on Listing 3.

Input generation strategies, life-cycle methods, rules, and auxiliary assertion methods are detailed on subsequent Listings.

Strategies and life-cycle methods

- Listing 4 highlights input generation strategies as well as the methods used to instantiate tokens, wallets, the contract, as well as define address ranges for each cycle - *i.e.*, __init__ and setup. The teardown method is also shown.

```
class StateMachine:
    st_owner = strategy("uint256", min_value=0, max_value=Options.ACCOUNTS - 1)
    st_sender = strategy("uint256", min_value=0, max_value=Options.ACCOUNTS - 1)
    st_receiver = strategy(
        "uint256", min_value=0, max_value=Options.ACCOUNTS - 1
    )
    st_token = strategy("uint256", min_value=1, max_value=Options.TOKENS)
    st_bool = strategy("bool" )

    def __init__(self, wallets, contract, DEBUG=None):
        self.wallets = wallets
        self.addr2idx = { addr: i for i,addr in enumerate(wallets)}
        self.addr2idx[0] = -1
        self.addr2idx[brownie.convert.to_address('0x0000000000000000000000000000000000000000')
    ] = -1
        self.tokens = range(1, Options.TOKENS + 1)
        self.contract = contract

    def setup(self):
        # tokenId -> owner address - must match contract's ownerOf(tokenId)
        self.owner = {tokenId: self.contract.address for tokenId in self.tokens}

        # address -> number of tokens - must match contract's balanceOf(address)
        self.balance = {addr: 0 for addr in range(Options.ACCOUNTS)}

        # tokenId -> approved address - must match contract's getApproved(address)
        self.approved = {tokenId: -1 for tokenId in self.tokens}

        # address -> list of approved operators - for each address x in operators[address]
        # isApprovedForAll(address,x) must return true
        self.operators = {addr: set() for addr in range(Options.ACCOUNTS)}

        # Callback for initial setup (contract-dependent)
        self.onSetup()
    . . .
    def teardown(self):
        if Options.DEBUG:
            print("teardown()")
            self.dumpState()
. . .
```

Listing 4: ERC-721 state machine - strategies for input generation and life-cycle methods (__input__ , setup, and teardown.

```
    def rule_transferFrom(self, st_owner, st_receiver, st_token, st_sender):
        if Options.DEBUG:
            print(
                "transferFrom(owner {},receiver {},token {} [sender: {}])".format(
                    st_owner, st_receiver, st_token, st_sender
                )
            )
        if self.canTransfer(st_sender, st_owner, st_token):
            tx = self.contract.transferFrom(
                self.wallets[st_owner],
                self.wallets[st_receiver],
                st_token,
                {"from": self.wallets[st_sender]},
            )
            self.owner[st_token] = st_receiver
            self.approved[st_token] = -1
            self.balance[st_owner] = self.balance[st_owner] - 1
            self.balance[st_receiver] = self.balance[st_receiver] + 1

            self.verifyOwner(st_token)
            self.verifyApproved(st_token)
            self.verifyBalance(st_owner)
            self.verifyBalance(st_receiver)
            self.verifyEvent(
                tx,
                "Transfer",
                {
                    "_from": self.wallets[st_owner],
                    "_to": self.wallets[st_receiver],
                    "_tokenId": st_token
                },
            )
        else:
            with brownie.reverts():
                self.contract.transferFrom(
                    self.wallets[st_owner],
                    self.wallets[st_receiver],
                    st_token,
                    {"from": self.wallets[st_sender]},
                )
    def rule_safeTransferFrom(self, st_owner, st_receiver, st_token, st_sender):
        ...
    def rule_approve(self, st_sender, st_token, st_receiver):
        ...
    def rule_setApprovalForAll(self, st_sender, st_receiver, st_bool):
        ...
```

Listing 5: ERC-721 state machine - rules.

Rules

- Listing 5 exemplifies one rule for testing the transferFrom function of ERC721. The methods starts by calling the auxiliary method canTransfer that checks whether the transfer is being emitted by the token's owner or by an approved operator for it, as well as if the token's current address (ownership) is that of the st_owner argument passed. It reverts the test otherwise identifying an error.

- The method then calls the contract function in question and inspects if the expected state changes on the blockchain took place. Namely, it internally updates the token's owner to be that of the receiver's address, removes third party approval for that token, and performs the corresponding updates to each account address' balance. It then verifies if those changes were successfully carried out by the blockchain, by calling a series of verification methods - transversal to other rule methods.

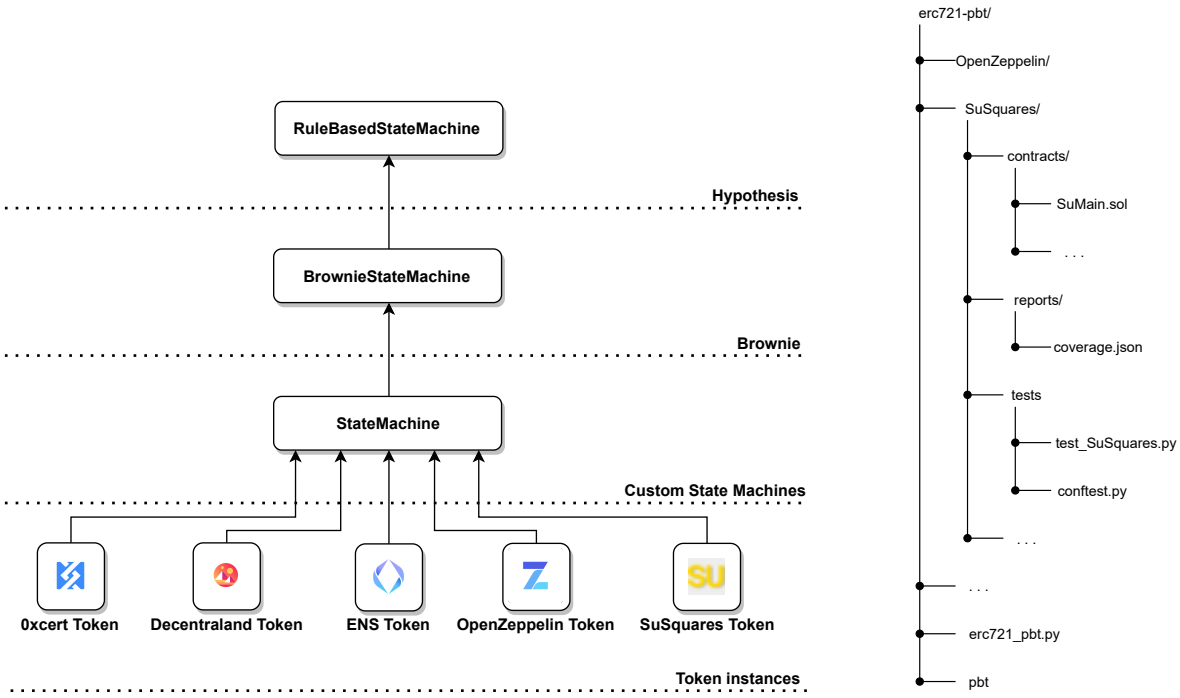
```
    def verifyOwner(self, tokenId):
        self.verifyValue(
            "ownerOf({})".format(tokenId),
            self.owner[tokenId],
            self.addr2idx[self.contract.ownerOf(tokenId)]
        )

    def verifyBalance(self, wIdx):
        self.verifyValue(
            "balanceOf({})".format(wIdx),
            self.balance[wIdx],
            self.contract.balanceOf(self.wallets[wIdx]),
        )

    def verifyApproved(self, token):
        self.verifyValue(
            "getApproved({})".format(token),
            self.approved[token],
            self.addr2idx[self.contract.getApproved(token)]
        )

    def verifySetApprovedForAll(self, sender, receiver):
        ...
    def verifyEvent(self, tx, eventName, data):
        ...
    def verifyReturnValue(self, tx, expected):
        ...
    def verifyValue(self, msg, expected, actual):
        ...
    def verifyValue(self, msg, expected, actual):
        if expected != actual:
            self.value_failure = True
            raise AssertionError(
                "{} : expected value {}, actual value was {}".format(
                    msg, expected, actual
                )
            )
    ...
```

Listing 6: ERC-721 state machine - overview of auxiliary methods.

Assertion Methods

- Sample verification methods can be examined on Listing 6. All of these methods follow equivalent steps, where they verify if the expected value matches the one read from the blockchain.

- The actual value verification method is verifyValue and, as straightforward as it is, was placed on a separate method simply for abstraction purpose, while writing the remaining auxiliary verification methods.



(a) Test Class Hierarchy.

(b) Directory structure.

Figure 6: Project organization.

3.3.2 Class Hierarchy and Project Organization

Test classes are hierarchically structured as depicted on Figure 6a. The infrastructure is built, as shown, supported on Hypothesis then Brownie, following the ERC-721 state machines of this project's PBT framework and, ultimately, test instantiations for the contracts.

The PBT framework of Hypothesis defines the `RuleBasedStateMachine` class, upon which Brownie adapts it into the `BrownieStateMachine` class. In this way, the general stateful testing framework of Hypothesis is polished into one that abstracts aspects related to blockchain setup, snapshots, and state rollbacks, in order to allow for test isolation.

Brownie, following its custom norms, defines rule-based state machines as specified by the code, and as this project's ERC-721 `StateMachine` serves as example. State machine classes can then be subclassed for contracts through the test scripts.

Directory structure for defining this PBT framework is schematized on Figure 6b.

At the root level sits the files `erc721_pbt.py` and `pbt`. The former contains the PBT implementation, while the latter is the bash script used for text execution. Each ERC-721 smart contract is given its own sub-directory, inside of which there is a Brownie project containing follow-up sub-directories that, of these, it is important to highlight a few. Namely, specific contract source code that is placed inside `contracts/`; test scripts, which can be found in `tests/`; and, finally, coverage reports and logs that are located in `reports/`.

4 Evaluation

4.1 Methodology

Table 1: Brief summary of the tested Ethereum ERC-721 contracts

Contract	Description
Decentraland [7]	Virtual reality platform where users can create, discover, experience, and monetize content and applications
Ethereum Name Service [11]	Domain Name service for addresses, including wallets and websites
Su Squares [20]	Provides a homepage with blocks, available for purchase, and containing a website link and picture attached
0xcert [1]	Reference Implementation
OpenZeppelin [16]	Reference Implementation

In order to ascertain the impact produced by the number of steps per test cycle, we carried out two batches of tests with a different number of steps each. At the lower end, only a maximum of five steps for each test cycle were made, while at the higher end a total of one hundred steps could be taken. Ten executions were performed for each series of tests with a given number of steps. Half of them with shrinking, and half without.

We conducted testing on five Ethereum ERC-721 smart contracts, written in Solidity, out of which three are real-world contracts and two are reference implementations. The tested contracts are outlined on Table 1.

4.2 Overall Results

Table 2: Overall results per contract with a max of 100 examples each.

			Time (s)		Time w/Shrinking (s)		
Contract	**Steps**	**# Bugs**	**Average**	**Deviation**	**Average**	**Deviation**	**Coverage** (%)
0xcert	5	1	155.6	2.7	160.2	2.5	70.2
0xcert	100	1	979.3	7.2	988.4	5.2	72.6
Decentraland	5	4	217.3	3.2	501.9	3.1	68.8
Decentraland	100	5	613.5	1.7	913.9	1.5	70.7
ENS	5	0	136.4	2.6	136.2	2.9	46.1
ENS	100	0	1457.1	4.6	1460.0	7.7	47.6
OpenZeppelin	5	0	135.7	1.2	136.1	2.7	80.9
OpenZeppelin	100	0	1311.3	29.4	1297.4	7.9	83.6
Su Squares	5	1	206.8	4.9	216.3	2.3	38.7
Su Squares	100	1	697.7	4.4	711.4	2.2	38.7

Table 2 lists the results for two sets of test executions for the five ERC-721 property-based tested contracts.

As was to be expected, a higher number of steps took longer to execute, but also yielded a better coverage and could expose further errors. As also anticipated, test executions with shrinking entailed

significant time overhead. This overhead is heavily dependent on the smart contract in question, no doubt due to how the ERC-721 standard is being implemented in code.

Pointing to this fact, is the discrepancy between the minor increase in time for two out of three contracts where bugs were found, and the major time increment that occurred with the third one - the Decentraland contract.

Of note that this time parameter holds no significance for smart contracts that did not exhibit errors, since shrinking only takes place when a test fails. Specifically, this is demonstrated by the averages for the Ethereum Name Service smart contract, which are almost similar. Otherwise, while still on this subject and pertaining to the OpenZeppelin contract, the higher time value obtained without shrinking - for one hundred steps - can be explained by taking into account the larger than usual deviance, responsible for skewing this average. In turn, this deviance increase is most likely due to fortuitous concurrent background processes, as well as system overload caused by performing a large amount of examples and steps for a long period of time.

Further testing could be done to obtain more representative values and with greater confidence, but the figures obtained already allow to draw these solid inferences. This is substantiated by the overall small deviation in execution times, that points to the fact that the results thus obtained are sufficiently robust and consistent across the board.

```
contract: SuMain — 38.7%
  SuNFT.getApproved — 100.0%
  SuNFT.isApprovedForAll — 100.0%
  SuNFT.setApprovalForAll — 100.0%
  SuPromo.grantToken — 100.0%
  SuNFT._transfer — 95.0%
  AccessControl.setOperatingOfficer — 75.0%
  SuNFT.balanceOf — 75.0%
  SuNFT.transferFrom — 70.8%
  SuNFT._safeTransferFrom — 50.0%
  SuNFT.ownerOf — 50.0%
  SuNFT.safeTransferFrom — 50.0%
  SuNFT.approve — 41.7%
  AccessControl.setExecutiveOfficer — 0.0%
  AccessControl.setFinancialOfficer — 0.0%
  AccessControl.withdrawBalance — 0.0%
  SuNFT.name — 0.0%
  SuNFT.symbol — 0.0%
  SuNFT.tokenByIndex — 0.0%
  SuNFT.tokenOfOwnerByIndex — 0.0%
  SuNFT.tokenURI — 0.0%
  SuOperation.personalizeSquare — 0.0%
  SuVending.purchase — 0.0%
  SupportsInterface.supportsInterface — 0.0%

contract: TokenReceiver — 0.0%
```

Listing 7: Coverage report for the Su Squares contract

Regarding coverage, once more there is a small increase - in most cases - to the contract code that is covered with PBT, the more number of steps it is performed. Still, upon analysis of the coverage logs, we can deduce that most of the contract code that is not run during testing belongs to methods

specific to each DApp itself, and are therefore themselves not covered by the ERC-721 standard. The coverage report shown on Listing 7 serves as an example to this. Furthermore, the higher coverage obtained for the standard implementations further substantiates this claim.

Still, further analysis and tweaking of the strategies and rules defined by our test code could provide for better coverage.

4.3 Bug Analysis

Table 3: Bug summary.

Contract	**Function**	**Summary description of function and detected error(s)**
0xcert	setApprovalForAll	Enable or disable 3rd party operator for approval. Transaction did not revert for sender = receiver (assertion error).
Decentraland	approve	Change or reaffirm the approved address for an NFT. Function threw exception for reaffirmation when it shouldn't (receiver already an approved operator). Function does not fire for multiple individual approve events.
Decentraland	safeTransferFrom	Safe version of tranferFrom, that changes ownership of NFT. Third party approved operator attempted to transfer NFT from holder to itself (owner = receiver).
Decentraland	setApprovalForAll	Enable or disable 3rd party operator for approval. Transaction did not revert for sender = receiver (assertion error). Threw exception when clearing approval from approved operator.
Su Squares	approve	Change or reaffirm the approved address for an NFT. Transaction did not revert for sender = receiver (assertion error).

A summary of all the bugs found, and brief description thereof, can be reviewed on Table 3.

It is clear that both the OpenZeppelin and the Ethereum Name Service smart contracts contain no apparent faulty implementations of the ERC-721 standard. On the other hand, one bug was found on the 0xcert and Su Squares contracts - of a different nature each.

Once again, the Decentraland smart contract stood out, by revealing to contain more errors than the remaining contracts.

Most of the falsifiable examples, which translates into bad implementations of the standard, are related to the methods for approval. With property-based testing we found one instance on the Su Squares smart contract where the transaction did not revert. For Decentraland, an error occurred when performing a larger number of steps that involved multiple approve events. There was also a test instance where the reaffirmation of an approved operator returned an exception.

Again, the bug that was discovered in the setApprovalForAll function of 0xcert was related to a failure to revert operator approval. It is relevant to indicate that all these failures pertaining to approval reversal occurred when the contracts in question did not account for the sender and the receiver being the same address. Decentraland also presented this bug.

The last bug that was found, and also on Decentraland, was on the safeTransferFrom method, when

an approved third party attempted to transfer a token to its already current owner.

```
Falsifying example:
state = BrownieStateMachine()
state.rule_setApprovalForAll(st_bool=True, st_receiver=3, st_sender=1)
state.rule_setApprovalForAll(st_bool=True, st_receiver=3, st_sender=2)
state.rule_transferFrom(st_owner=2, st_receiver=1, st_sender=2, st_token=5)
state.rule_setApprovalForAll(st_bool=True, st_receiver=2, st_sender=1)
state.rule_approve(st_receiver=0, st_sender=3, st_token=6)
state.teardown()
Traceback (most recent call last):

  File "/home/halcyon/erc721-pbt/erc721_pbt.py", line 162, in rule_approve
  self.wallets[st_receiver], st_token,{"from": self.wallets[st_sender]}
  brownie.exceptions.VirtualMachineError: revertTrace step -1, program counter 1601:
  File "contracts/ERC721Base.sol", line 172, in ERC721Base.approve:
     */
    function approve(address operator, uint256 assetId) external {
      address holder = _ownerOf(assetId);
      require(msg.sender == holder || _isApprovedForAll(msg.sender, holder));
      require(operator != holder);
      if (_getApprovedAddress(assetId) != operator) {
```

Listing 8: Example of a PBT output, for the Decentraland contract, detailing one found bug

```
Falsifying example:
state = BrownieStateMachine()
state.rule_approve(st_receiver=0, st_sender=0, st_token=1)
state.rule_setApprovalForAll(st_bool=True, st_receiver=0, st_sender=1)
state.rule_approve(st_receiver=0, st_sender=0, st_token=1)state.teardown()
Traceback (most recent call last):
File "/home/halcyon/erc721-pbt/erc721_pbt.py", line 162, in rule_approve
self.wallets[st_receiver], st_token,{"from": self.wallets[st_sender]}
brownie.exceptions.VirtualMachineError: revertTrace step -1, program counter 1601:
  File "contracts/ERC721Base.sol", line 172, in ERC721Base.approve:
     */
    function approve(address operator, uint256 assetId) external {
      address holder = _ownerOf(assetId);
      require(msg.sender == holder || _isApprovedForAll(msg.sender, holder));
      require(operator != holder);
      if (_getApprovedAddress(assetId) != operator) {
```

Listing 9: The same bug as that of Listing 8, now with shrinking

Finally, regarding the outputs of the tests, and on the subject of shrinking, Listing 8 illustrates the section relevant to a falsifiable example, for a typical test output. As shown, the output indicates the generated inputs and rules for which the bug could be reproduced, as well as the covered code on the smart contract itself.

Listing 9 gives the output for the same bug but with shrinking. As can be observed, shrinking significantly reduces the size of the output, in its attempt to find a smallest reproducible failing example and improve readability, thus making it easier to detect the cause for the error in question.

For this particular case, by evidencing that the bug happened for an approval reaffirmation, shrinking helps identify that the error is due to the call of function _isApprovedForAll having the msg.sender and

holder arguments swapped. Since this was a reaffirmation, and an authorized third party is allowed to emit approvals, Brownie was not expecting an exception to be fired on that line of code. On the given function, instead of verifying if the approval was being sent by an authorized operator, it was checking if the owner of the asset in question was instead an operator on behalf of the sender - who is the actual operator.

5 Conclusion

In this project, we presented a PBT framework for ERC-721 contracts. The model, relying on rule-based state machines, was built on top of Brownie development framework, and made possible by its incorporation of the Hypothesis library for PBT. Evaluations were conducted on five different Ethereum non-fungible token smart contracts, out of which three are real-wold contracts - specifically Decentraland, Ethereum Name Service, and Su Squares - and two are reference implementations - 0xcert and OpenZeppelin.

We reached the following conclusions, as highlighted below:

- Two of the three real-world implementations contained bugs, the Su Squares and the Decentraland ones, with the latter displaying a significant larger amount than all the rest. One reference implementation, 0xcert, also contained a bug. This provides evidence that several ERC-721 contracts may exhibit bugs and discrepancies to the standard.
- The PBT approach was able to obtain consistent results throughout the several test execution for each contract, proving to be as sturdy an approach as unit testing might have been, while allowing for the generation of hundreds of examples across multiple tests and free from the constraint of the programmers insight regarding the code.
- Most of the bugs that were found pertained, in one way or another, with the methods for third-party operator approval. This could hint that the most lacking aspect of ERC-721 on current implementations might still be smart contract interaction with external services.
- That the Decentraland smart contract presented an error on a safe transfer method, not only does not bode well for it, as it alerts to the possibility that many other existing contracts may contain similar or even worse faulty implementations of vital functions.

With our findings, we can generally conclude that property-based testing constitutes a promising approach for exposing bugs in ERC-721 contracts, and potentially other types of Ethereum smart contracts.

References

[1] 0xcert erc-721 token — reference implementation - github repository. https://github.com/0xcert/ethereum-erc721. [Accessed: September 2020].

[2] Brownie - property-based testing. https://eth-brownie.readthedocs.io/en/stable/tests-hypothesis-property.html. Accessed: 2021-02-08.

[3] Vitalik Buterin. Ethereum whitepaper. https://ethereum.org/en/whitepaper/, 2013. [Online; accessed 01/02/2021].

[4] Usman W. Chohan. The double spending problem and cryptocurrencies. *SSRN*, 2017.

[5] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. ICFP'2000*. ACM, 2000.

[6] CryptoKitties collect and breed digital cats. https://www.cryptokitties.co/. Accessed: 2021-02-08.

[7] Decentraland erc 721 - github repository. https://github.com/decentraland/erc721. [Accessed: September 2020].

[8] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. Eip-721: Erc-721 non-fungible token standard. https://eips.ethereum.org/EIPS/eip-721. Accessed: 2021-02-08.

[9] Erc-721 property-based testing. https://github.com/RoninKingfisher/erc721-pbt. [Accessed: March 2021].

[10] Ethereum improvement proposals. https://eips.ethereum.org/. Accessed: 2021-02-08.

[11] Ens: Base registrar implementation. https://etherscan.io/address/0x57f1887a8bf19b14fc0df6fd9b2acc9af147ea85#code. [Accessed: September 2020].

[12] Ganache. https://github.com/trufflesuite/ganache-cli. [Accessed: September 2020].

[13] Hypothesis documentation. https://hypothesis.readthedocs.io/en/latest/index.html. Accessed: 2021-02-08.

[14] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[15] Rickard Nilsson. *ScalaCheck: The Definitive Guide*. Artima, 2014.

[16] Openzeppelin - github repository. https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v2.2.0. [Accessed: September 2020].

[17] OpenZepplin - ERC 721. https://docs.openzeppelin.com/contracts/2.x/api/token/erc721.

[18] Célio Rodrigues. Property-based testing of ERC-20 smart contracts. Master's thesis, Universidade do Porto, 2020.

[19] The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity.

[20] Su squares ethereum contract - github repository. https://github.com/su-squares/ethereum-contract. [Accessed: September 2020].

[21] Fabian Vogelsteller and Vitalik Buterin. ERC-20 token standard, 2015. Ethereum Foundation – https://eips.ethereum.org/EIPS/eip-20.

[22] Vyper: Pythonic Smart Contract Language for the EVM. https://github.com/vyperlang/vyper.

[23] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Foundation, 2014.