Checking Laws of the Blockchain With Property-Based Testing

Alexander Chepurnoy*, Mayank Rathee†
* Ergo Platform and IOHK Research
Sestroretsk, Russia
† Department of Computer Science and Engineering, Indian Institute of Technology (Banaras Hindu University)
Varanasi, India

***Abstract*—Inspired by the success of Bitcoin, many clients for the Bitcoin protocol as well as for alternative blockchain protocols have been implemented. However, implementations may contain errors, and the cost of an error in the case of a cryptocurrency can be extremely high.**

We propose to tackle this problem with a suite of abstract property tests that check whether a blockchain system satisfies laws that most blockchain and blockchain-like systems should satisfy. To test a new blockchain system, its developers need to instantiate generators of random objects to be used by the tests. The test suite then checks the satisfaction of the laws over many random cases. We provide examples of laws in the paper.

I. Introduction

A blockchain-based cryptocurrency system consists of a set of policies and protocols, such as the consensus protocol, the monetary policy for token emission, the rules for transaction processing, a peer management protocol and network communication protocols. A *node*, or a *client*, is a software implementation of the protocols. Usually not all the details of all the protocols are rigorously specified. Instead, typically there is a reference client implementation that acts as the definition of the protocols for other implementations. A notable exception here is Ethereum, where the Yellow Paper [1] tries to define all the details of a client implementation. In Bitcoin, however, the reference implementation Bitcoin Core is considered to be standard, and any alternative implementation is expected to reproduce its behavior, and even its bugs [2].

Repeated testing of even the most carefully written and designed system is crucial to expose hidden vulnerabilities in the developed system which might miss the eye of the developers. Such tests should be performed regularly in order help ensure reliability, security and performance of the system. Furthermore, in the case of Bitcoin and other cryptocurrencies, an error in a client implementation could be utterly costly and hard to fix. For example, the famous value overflow bug in Bitcoin [3] caused a fork of more than 50 blocks (more than 8 hours) and required a soft fork (for the majority of miners to upgrade) to be fixed. With an increasing demand for the development of more clients for existing as well as new alternative blockchain systems and cryptocurrencies, such costly bugs can be expected to become more common and problematic and testing can be expected to become one of the most important parts of the software development lifecycle for blockchain systems.

This paper addresses these trends by:

1) proposing a generic property-based testing framework that can be easily plugged into the implementation of a concrete client;
2) describing some of the essential properties which ought to hold true for any cryptocurrency implementation.

A. Property-Based Testing

In this section, we give a formal definition of a property followed by a discussion on property-based testing in contrast to conventional testing methodologies.

Within the scope of a data domain $\mathbb{D}$, a property can be seen as a collective abstract behavior which has to be followed by every valid member of the data domain. More precisely, a property is a predicate $P : \mathbb{D} \rightarrow \{true, false\}$ and it is desirable that it be *valid*:

$$\forall X \in \mathbb{D}, P(X) = true$$

To illustrate, an example of a property P over the domain of pairs of strings $\mathbb{S} \times \mathbb{S}$ is shown below:

$$P((s_1, s_2)) = \#(s_1 :: s_2) > \#s_1$$

where :: denotes string concatenation and $\#s$ denotes the length of string s. This property is false for any (s_1, ε), where ε is the empty string. Therefore, it is not valid.

In contrast to conventional testing methods, where the behavior of a program is only tested on some pre-determined cases, property-based testing [4] emphasizes defining properties and then testing their validity against randomly sampled data points. As property-based testing uses a small number of randomly sampled data points, it still provides only an approximate answer to the question of whether a property is satisfied on all data points. However, it may provide more confidence than conventional unit testing, because the randomly sampled data points may cover problematic cases that were not foreseen by the developers. There are various popular libraries available for property testing including QuickCheck for Haskell [5], JUnit-QuickCheck for Java [6], theft for C, ScalaTest [7] and ScalaCheck [8] for Scala.

Property-based testing is also advantageous when testing an application developed on top of a general framework, as is the case of blockchain systems developed on top of Scorex, because the framework may provide pre-implemented properties that the application should satisfy and the application developer just needs to implement application-specific generators of random data points.

B. The Scorex Framework

The idea of a modular design for a cryptocurrency was first proposed by Goodman in the Tezos position paper [11]. The paper (in Section 2) proposes to split a cryptocurrency design into the three protocols: network, transaction and consensus. In many cases, however, these layers are tightly coupled and it is hard to describe them separately. For example, in a proof-of-stake cryptocurrency a balance sheet, which representation is heavily influenced by a transaction format, is used in a consensus protocol.

Plenty of modular open-source frameworks were proposed for speeding up development of new blockchain systems, including: Sawtooth [12] and Fabric [13] by Hyperledger, Exonum [14] by Bitfury Group, and Scorex [15] by IOHK. We have chosen Scorex, because it has finer granularity. In particular, in order to support hybrid blockchains as well as more complicated linking structures than a chain (such as Spectre[16]), Scorex does not have the notion of blockchain as a core abstraction. Instead, it provides a more general abstract interface to a *history* which contains *persistent modifiers*[1]. The history is a part of a *node view*, which is a quadruple of $\langle$*history*, *minimal state*, *vault*, *memory pool*$\rangle$. The node view is updated whenever a persistent modifier or a transaction is processed. The minimal state is a data structure and a corresponding interface providing the ability to check the validity of an arbitrary persistent modifier for the current moment of time with the same result for all the nodes in the network having the same history. The minimal state is to be obtained deterministically from an initial pre-historical state and the history. The vault holds node-specific information, such as a user's wallet. The memory pool holds unconfirmed transactions being propagated across the network by nodes before their inclusion into the history. Such a design, described in details in Section II, gives us the possibility to develop an abstract testing framework where it is possible to state contracts for the node view quadruple components without knowing details of their implementations.

C. Our Contribution

This paper reports on the design and implementation of a suite of abstract property tests which are implemented on top of the Scorex framework to ease checking whether a blockchain client satisfies the specified properties. A developer of a concrete blockchain system just needs to implement generators of random test inputs (for example, random blocks and transactions for a Bitcoin-like system), and then the testing system will extensively check properties against multiple input objects. We have implemented 59 property tests, and integrated them into a prototype implementation of the TwinsCoin [17] cryptocurrency.

[1]In a blockchain-based cryptocurrency, the blockchain can be seen as the history and every block can be seen as a persistent modifier.

D. Related Work

Verification and testing of software systems [18] is an integral part of a software development lifecycle. Immediately after the implementation of the software, and before its deployment, it has to be verified and tested extensively enough to ensure that all the functional requirements have been properly met. Over the course of time, both testing and verification methods have been becoming increasingly formal, sophisticated and automated.

Formal verification usually involves constructing an abstract mathematical model (a.k.a. specification) of the system's desired behavior. From a logical perspective, the specification can be regarded as a collection of properties that ought to hold for the system, although often the specification is not described directly in logical form, but rather using various mathematical modeling frameworks, such as finite state machines [19], Petri Nets, process algebra and timed automata [20]. Once both the specification and the system are ready, the actual verification that the program satisfies the specification can be attempted. If successful, the verification proves that the properties of the specification are valid for the program. This is a starkly stronger result than what can be achieved through testing, where the properties are only checked on a few samples. However, full verification is hard to achieve automatically, and expensive to do manually or interactively.

Testing (either conventional or property-based) remains a less costly and hence more prevalent approach. Since a software program is developed at module or class level and is integrated with other modules or classes along the development cycle, testing is done at unit level, integration level and system level [18], before the software is deployed. End-to-end testing [21] is also performed, usually after system testing, to validate correct flow spanning different components of the software in real world use cases. Unit tests target individual modules, methods or classes and have a small coverage compared to integration tests which aim towards checking the behavior of modules when combined together. The two main approaches to unit testing are black box testing and white box testing. The former one focuses on designing test instances without looking inside the code or design, in other words, the black box testing focuses only on the extensional functionality of the unit under testing, while the white box testing approach is more inclined towards code coverage (i.e. ensuring that test instances execute as many different paths of the code as possible).

Although initially white box testing was considered only as a method for unit testing, recently it has emerged as a popular method for integration testing as well. Integration testing is usually done by one or a combination of the following approaches:

1) *Big-Bang approach:* all the components are integrated together at once and then tested. This method works well for comparatively smaller systems, but is not well suited for larger systems. One obvious disadvantage is that the testing can only begin after all the individual components have been built.

2) *Top-Down approach:* the modules at upper level are tested first and then we move down until we test the lowest level modules at the end. Since lower level modules might not be developed when the upper ones are being tested, stubs are used in place of such modules. The stubs try to simulate behaviour of the modules not yet implemented.
3) *Bottom-Up approach:* in contrast to the top-down approach, here the lower level modules are tested first and then we iteratively move upwards in the hierarchy until we reach the highest level module. Now as we are testing lower level modules first, stubs are used to simulate the behaviour of not yet implemented higher level modules, in case any sibling interaction is required.
4) *Sandwich approach:* this combines the Bottom-Up and Top-Down approaches.

Going beyond conventional unit testing methods, which do not take any input parameters, parameterized unit tests [22] are generalized tests that have an encapsulated collection of test methods whose invocation and behaviour is controlled by a set of input parameters giving more flexibility and automation to unit testing as a whole.

The final full scale testing that a software product undergoes is called the system testing, which includes tests like security test, compatibility test, exceptions handling, scalability tests, stress tests and performance tests.

Stress tests are particularly important for electronic payment systems, even conventional ones that are not based on cryptocurrencies. Visa, for instance, performed an annual stress test in 2013 to prepare their VisaNet system for the peak traffic of the upcoming holiday season. The test results showed that the system was able to handle 47,000 transactions per second, a 56% improvement compared to the system's capacity in the previous year. Within cryptocurrencies, the Bitcoin network experienced a spam campaign called *"stress test"* [23], which caused the network's performance to degrade and essentially resulted in a denial-of-service attack [2]. The intention behind this campaign was to expose vulnerabilities of the network, particularly when facing spam attacks. The maximum transaction verification rate of a network under spam can be improved through clustering of transactions to differentiate spam and genuine transactions [23] or through $UTXO$-cleanup transactions, a new special type of transaction created by miners to combine spam transactions together, thereby reducing the $UTXO$ set size and the impact of the spam attack on the network.

E. Structure of the Paper

In Section II we explain the architecture of the Scorex framework. We then describe our approach to property-based testing of blockchain system properties and present many examples of blockchain property tests in Section III. Finally we state our conclusions in Section IV.

[2]a cyber-attack on a system where the attack makes the system's resources unavailable or degrades their quality to a point where it becomes difficult or sometimes impossible for honest users to avail the resource

II. Scorex Architecture

Scorex is a framework for rapid implementation of a blockchain protocol client. A client is a node in a peer-to-peer network. The client has a local view of the network state. The goal of the whole peer-to-peer system [3] is to synchronize on a part of local views which is a subject of a consensus algorithm. Scorex splits a client's local view into the following four parts:

- *history* is an append-only log of *persistent modifiers*. A modifier is persistent in the sense that it has a unique identifier, and it is always possible to check if the modifier was ever appended to the history (by presenting its identifier). There are no limitations on a modifier structure, besides the requirements to have a unique identifier and at least one parent (referenced by its identifier). A persistent modifier may contain transactions, but this is optional. A transaction, unlike a persistent modifier, has no mandatory reference to its parents; also we consider that a transaction is not to be applied to the history and a minimal state (described below). If a modifier is applicable to a history instance and so could be appended to it, we say that the modifier is *syntactically* valid. As an example, in a Bitcoin-like blockchain the history is about a chain of blocks. A block is syntactically valid if its header is well-formed according to the protocol rules, and current amount of work was spent on generating it. However, a syntactically valid block could contain invalid transaction, see a notion of semantic validity below. We note that there are alternative blockchain protocols with multiple kinds of blocks, microblocks, paired chains, and so on, that is why we have chosen abstract notions of a persistent modifier and a history, not the block and the blockchain.
- *minimal state* is a structure enough to check semantics of an arbitrary persistent modifier with a constraint that the procedure of checking has to be deterministic in nature. If a modifier is valid w.r.t minimal state, we call it a *semantically* valid modifier. Thus, in addition to syntax of the blockchain, there is some stateful semantics, and minimal state takes care of it. That is, all nodes in the system do agree on some pre-historical state S_0, and then by applying the same sequence of persistent modifiers $m_1, \ldots, m_k$ in a deterministic way, all the nodes get the same state $S_k = apply(\ldots apply(apply(S_0, m_1), m_2), m_k)$ if all the $m_1, \ldots, m_k$ are *semantically* valid; otherwise a node gets an error on the first application of a semantically invalid persistent modifier. From this more abstract point of view, the goal of obtaining the state S_k is to check whether a new modifier m_{k+1} will be valid against it or not. Thus the minimal state has very few mandatory functions to implement, such as $apply(\cdot)$ and $rollback(\cdot)$ (the latter is needed for forks processing).

[3]concretely, its honest nodes. For simplicity, we omit a notion of adversarial behavior further.

- *vault* contains user-specific information. For a user running a node, the goal to run it is usually to get valuable user-specific information from processing the history. For that, the vault component is used which has the only functions to update itself by scanning a persistent modifier or a transaction and also to rollback to some previous state. A wallet is the perfect example of a vault implementation.
- *memory pool* is storing transactions to be packed into persistent modifiers.

The history and the minimal state are parts of local views to be synchronized across the network by using a distributed and decentralized consensus algorithm. Nodes run a consensus protocol to form a proper history, and the history should result in a valid minimal state when persistent modifiers from the history are applied to a publicly known prehistorical state.

The whole node view quadruple is to be updated atomically by applying either a persistent node view modifier or an unconfirmed transaction. Scorex provides guarantees of atomicity and consistency for the application while a developer of a concrete system needs to provide implementations for the abstract parts of the quadruple as well as a family of persistent modifiers.

A central component which holds the quadruple <*history, minimal state, vault, memory pool*> and processes its updates atomically is called a *node view holder*. The holder is processing all the received commands to update the quadruple in sequence, even if they are received from multiple threads. If the holder gets a transaction, it updates the vault and the memory pool with it. Otherwise, if the holder gets a persistent modifier, it first updates the history by appending the modifier to it. In a simplest case, if appending is successful (so if the modifier is syntactically valid), the modifier is then applied to the minimal state. However, sometimes a fork happens, so the state is needed to be rolled back first, and then a new sequence of persistent modifiers is to be applied to it.

As an example, we consider the cryptocurrency Twinscoin [17], which is based on a hybrid proof-of-work and proof-of-stake consensus protocol. Scorex has a full-fledged Twinscoin implementation as an example of its usage. There are two kinds of persistent modifiers in Twinscoin: a proof-of-work block and a proof-of-stake block. Thus the blockchain is hybrid: after a Proof-of-Work block it could be only a Proof-of-Stake block, and on top of it there could be only a Proof-of-Work block again. Thus a TwinsCoin-powered blockchain is actually two chains braided together. Only Proof-of-Stake block could contain transactions. Such complicated design makes Scorex a good framework to implement the TwinsCoin proposal. Unfortunately, TwinsCoin authors made only some particular tests. We got working tests for the Twinscoin client just by writing generators for transactions and persistent modifiers.

A. Forking

It could be the case that in a decentralized network two generators are issuing a block at the same time, or in the presence of a temporary network split different nodes are working on different suffixes starting with the same chain, or an adversary may generate blocks in private and then present them to the network. In short, a fork could happen. This is a normal situation once majority of block generators are honest (see [24] for formal analysis of the Bitcoin proof-of-work protocol).

Processing forks in a client could be a complicated issue, making testing of this functionality important. We proceed by describing the way in which forking is implemented in Scorex. When a persistent modifier is appended to a history instance, the history returns (if the modifier is syntactically valid) *progress info* structure which contains a sequence of persistent modifiers to apply as well as a possible identifier of a modifier to perform rollback (for the minimal state, vault, memory pool) to before the application of the sequence. By such a realization of the interfaces, Scorex allows history to be non-linear (for example, it could be a block tree), but other components of the node view quadruple have sequential logic. For efficiency reasons, the minimal state is usually limited in maximal depth for a rollback, so the rollback could fail (this situation is probably unresolvable in a satisfactory way without a human intervention).

III. Property-based testing of a blockchain client

In this section we report on our approach to generalized exhaustive testing of an abstract blockchain (or blockchain-like) protocol implementation. For extensive testing, we test history, minimal state and memory pool components separately, and also do thorough checks for node view holder properties.

In total, we have implemented 59 property tests. They are using random object generators described in Section III-A. Most of the tests are relatively simple, others could check complex functionalities where several components are involved. We provide many examples in Section III-B.

A. Generators

We recall that (unlike unit tests, for example), property-based tests are checking not an output of a functionality under test against a concrete input, but rather a relation between input and output values for an arbitrary input value. Thus, in order to run a property-based test, an instance of an input value is needed. To be able to obtain it, a property-based test is supplied with a random input generator, which provides a random input domain element upon request. For our testing framework, a developer of a concrete protocol client needs to provide implementation for generators of the following types:

- a syntactically valid (respectively, invalid) modifier, which is valid (respectively, invalid) against given history instance
- a semantically valid (respectively, invalid) modifier, which is valid (respectively, invalid) against given minimal state instance. The modifier could be syntactically invalid

- a totally, so both semantically and syntactically, valid modifier. Respectively, a sequence of totally valid modifiers
- a transaction
- history instance, for which it should be possible to generate a syntactically valid modifier
- minimal state instance, for which it should be possible to generate a semantically valid modifier
- vault instance
- node view holder instance, for which it should be possible to generate a totally valid modifier

As an example, for the TwinsCoin implementation we provide concrete implementations for all the generators mentioned above. To generate a syntactically valid modifier, we generate a Proof-of-Work block if a previous pair of <*Proof-of-Work block, Proof-of-Stake block*> is complete, otherwise we generate a new Proof-of-Stake block. We recall that in TwinsCoin transactions can be recorded only in Proof-of-Stake blocks. A minimal state in the TwinsCoin implementation, similarly to Bitcoin, is defined as a set of current unspent transaction outputs. In order to generate a semantically valid modifier, we generate a Proof-of-Stake block including transactions based on unspent transaction outputs. A totally valid modifier generator, based on given history as well as minimal state instances, produces either an empty Proof-of-Work [4] or a semantically valid Proof-of-Stake block, depending on the last block in the history (in order to generate the modifier which is also syntactically valid).

Interestingly, we implicitly define some properties via generators. In particular, the existence of a generator for a totally valid modifier for any given correct history and valid minimal state instances assumes that it is always possible to make a progress in constructing a blockchain. To the best of our knowledge, the need of this property to be hold was first stated in the formal model of the Bitcoin protocol by Garay et. al. [24] (see "Input Validity" definition in Section 3.1 of the paper [24]).

B. Examples of properties tests

To explain our approach to the testing of a client in details, in this section we provide some examples of property tests which are valid for most blockchain-based systems. We have grouped the tests based on their similarity.

1) *Memory pool Tests.*
   Memory Pool (or just mempool in the Bitcoin jargon) is used to store unconfirmed transactions which are to be included into persistent modifiers. The following tests are used to check some general properties of a memory pool which every blockchain client should pass.
   - *A memory pool should be able to store enough transactions.*

     In TwinsCoin implementation, we are testing that the memory pool which is empty before the test should be able to store a number of transactions up to a maximum specified in settings.

   - *Filtering of valid and invalid transactions from a memory pool should be fast.*

     We got an impression from running the Twinscoin client that memory pool probably spends too much time on filtering out a transaction. To be certain about that we have implemented a test which is checking that an implementation of memory pool is able to filter out a transaction reasonably fast. As processing time is platform-dependent, the test during its instantiation is measuring time to calculate 500,000 blocks of SHA-256 hash. Time to filter out the transaction should be no more than that. We found that the Twinscoin implementation was really inefficient about filtering.

   - *A transaction successfully added to memory pool should be available by a transaction identifier.*
     The purpose of this test is to ensure that once a transaction is added to the memory pool, it indeed is available by a transaction identifier. The test simply adds the transaction to the memory pool and then query the transaction by its identifier. The initial transaction is the only correct result of the compound operation.

2) *History Tests.*

   A history is an abstract data structure which records all the persistent modifiers ever appended to it. We recall that the blockchain structure in the Bitcoin protocol is the example of a history implementation. Since history is an integral part of a node view, it is important to check if an implementation of history acts correctly. A consensus protocol aims at establishing common history for all the nodes on the network.
   A persistent modifier is the main building block of a history and is used to update the history and a minimal state. As soon as a valid modifier got appended to history, the whole node's local view is being changed in the sense that the history is updated, possibly along with the minimal state.
   We have many tests to test history, some examples are provided below.
   - *A syntactically valid persistent modifier should be applicable to a history instance and available by its identifier after that.*
     By definition, a syntactically valid persistent modifier should be applicable to a history instance, and then, once applied to the history, it should be

[4]unlike Bitcoin, Twinscoin does not have a notion of a coinbase transaction rewarding miner, instead, block generator's public key is included into the block directly.

available by its unique identifier. The importance of this test comes from the fact that it is of utmost importance for the client implementation to tell the difference between the modifiers that have been appended to the history from those that have not been added. For this purpose, the unique identifier of the modifier can be used to query the history to know whether the modifier has been added to the history of not.

- *A syntactically invalid modifier should not be able to be added to history.*

  A syntactically invalid modifier should not be applicable to a history instance. The test first checks whether application of the modifier returns an indication of an error. Then the test checks that the modifier should not be available by its identifier.

- *Modifier not ever appended to a history instance should not be available by request.*
  When the persistent modifier which was never appended to the history, is queried from the history, it should always return empty result which shows that the invalid modifier has not been added to the history.

- *After application of a syntactically invalid modifier to a history instance, it should not be available in history by its identifier.*
  A syntactically invalid modifier is one which is inconsistent with the present view of history. Only syntactically valid modifier is eligible to be applied to history. To check how the history is filtering out invalid modifiers, we propose this test. We generate a random invalid modifier and attempt to add it to history. Since it is invalid, history should not add it and hence, it should not be available by identifier when queried from history. Some examples of generated invalid modifiers are ones with false nonces which do not satisfy the puzzle and ones with non-valid signatures.

- *Once a syntactically valid modifier is appended to history, the history should contain it.*
  This test ensures that if valid modifiers are correctly appended to the history, then they should be then available by their respectable identifiers. Also, the test is checking that the history is indicating success during the application.

- *History correctly reports semantic validity of an identifier.*

  A history instance should be able to indicate semantic validity of a persistent modifier. If the modifier is not appended to the history yet, the history should return on request that semantic validity of the modifier is not known. The same result should be returned once the modifier is appended, but semantic validity status is not provided by the node view holder (after applying the modifier to the minimal state). Once semantic validity status is provided, the history should return it by request. The test checks all the options, simulating node view holder with a stub.

3) *Minimal State Tests.*

Tests for the minimal state component are checking application of a semantically valid (respectively, invalid) persistent modifier, and also rollback functionality. In case of a better version of history (a fork) found, a rollback has to be performed which essentially rolls the system back to a common point (from which forks are started). Known examples of rollbacks performed in the Bitcoin network are recovery from the SPV mining issue [25], and also recovery from the arithmetic overflow bug [3].

- *Application and rollback should lead to the same minimal state.*

  In this test, a semantically valid persistent modifier m is generated and applied to a current state S. Due to this application of the modifier, a new minimal state S' is to be obtained from S. After the modifier is applied successfully to the history, a rollback is performed to take the system back to state S from the current state S'. The test now checks that the state to which the system comes after the rollback is indeed the state S by checking an identifier of the new state after rollback is the same as the identifier of the original state.

  We now use this test to explain how we generate a semantically valid modifier for the Twinscoin client. Before proceeding, we define the structure of a transaction t. A simple transaction is usually represented as the map $T : UTXO \rightarrow UTXO$, where $UTXO$ is the set of all the unspent transaction outputs or *boxes*. A box can be considered as a tuple $(pubkey, amount)$ where $pubkey$ is the public key of the account of the node to which this box belongs to and $amount$ refers to the monetary (in case of cryptocurrencies) amount which this box holds in the name of the $pubkey$ in the first half of the tuple. A transaction uses some ($\geq$ 0, 0 in the case of the rewarding transaction which is present at the end of each block and which rewards the miner) unspent boxes and generates new boxes with a constraint that sum of the amount of all the boxes used in the transaction is equal to the sum of

amount of the boxes output by the transaction except the rewarding transaction for which this constraint doesn't apply. Once the new boxes have been added to the UTXO set, the old boxes which were input to the transaction are removed from the UTXO set to prevent double spending. This addition and removal of boxes from $UTXO$ set has to be done atomically in order to avoid inconsistencies in the system. Suppose a node A wants to send x amount to a node B, then the transaction for this purpose will use some boxes with As public key on them which sum up to an amount $y \geq x$ and output the boxes b_1 and b_2 such that $b_1 = (B, x)$ which has Bs public key and an amount x whereas $b_2 = (A, y-x)$ will have As public key and an amount of $y - x$, if $y > x$. Now B has received a new box b_1 which belongs to him and this box now sits inside the set $UTXO$ until the point when B uses this box an one of the inputs to a future transaction. Readers should note that for more readability we will represent a transaction T by a tuple ($[in_1, in_2, ...]$, $[o_1, o_2, ...]$) where in_i denotes input boxes to the transaction and o_i denotes the output boxes of the transaction.
Along with checking that the ids are same, it also checks that the components of the new state are also rolled back and not just the id number got rolled back. For this, we generate the modifier m in the following way:

- Generate a pair of transactions (t_1, t_2) where $t_1 = ([b_1], [b_2])$ and $t_2 = ([b_2], [b_3])$. This notation means that the first transaction t_1 uses a box b_1 as its input and then outputs a box b_2 which is then used by the second transaction as its input. In the above setting, we select the first input box b_1 randomly from the set $UTXO$ and finally output the box b_3 from t_2 which also just generates a random box (b_3). The generated transaction pair has to be valid in the sense that it should only use valid unspent boxes from $UTXO$ and satisfy the constraint that the sum of amounts of all the input boxes should be equal to the sum of amounts of the output boxes. The main caveat here is that the second transaction of the pair should use the output box of the first transaction of the pair.
- Now we generate a pair of modifiers (m_1, m_2) and include both of these transactions from the pair above in the respective modifiers.

Once the custom modifiers are generated, m_1 (first half of the pair) is appended to the history and the system moves from state s_1 to the state s_2. As mentioned before, the transaction t_1 from the pair uses a random box b_1 from the $UTXO$ of state s_1 and when the system moves to the state s_2, the $UTXO$ gets added with the box b_2 and b_1 is removed from the set. Once the state change happens, we append m_2 (second half of the modifier pair) to history progressing the system to state s_3. Since m_2 contains the transaction t_2 which takes as input b_2, when the system moves to s_3, b_2 is removed from $UTXO$ and b_3 is added.

Now it becomes clear why we generate pairs of transactions and modifiers in the way defined above. Finally, we perform a rollback from state s_3 to s_2 which should mean that once the rollback is successful, the box b_2 should come back to the set $UTXO$ and should be available by id whereas the box b_3 should now not be present inside the $UTXO$ set anymore. Both of these checks tell us that the rollback was performed correctly and the system indeed came back to the previous state with all its components. The reason that we generated the pairs of transactions above is because it helps us in easily checking by id if b_2 has returned to the $UTXO$ set since we generated b_2 ourselves and know its id already. This makes testing easy and transparent.

- *Application of a valid modifier again after a rollback should be successful.*

  As the previous test aimed at checking that the components of a state are recovered after a rollback happens, it would be quite wrong to think that it should be the only test that is necessary to check if the rollback system performs as expected. It is also equally important that after rollback the system performs normally, as it would perform if the rollback would have never happened. To check this property to certain degree, we propose this test. It checks that after the rollback has happened the system becomes stable again and any new valid modifier which is now added to the history is actually recorded and hence should be available if queried from history. This test ensures that after recovering from a rollback the system performs normally and can resume its functioning without any issues. It hence ensures that a continuity is maintained after a rollback.

- *Double application of a semantically valid modifier should not be possible.*

  This test checks that a semantically valid persistent modifier should not be added more than once. For example, if in Bitcoin a block could be successfully applied twice to the validation state (UTXO set), all the transaction inside the modifier will be double spent. We argue that an implementation of a blockchain system should prevent addition of a semantically valid persistent modifier twice in general. In this test, we generate a semantically valid modifier, then append it to a generated minimal state once, and on the second application of the modifier again to the minimal state an error should be returned.

4) *Node View Holder Tests.*

As was mentioned in Section II, the node view holder is the central component of a blockchain client which is responsible for atomically updating the quadruple <*history, minimal state, vault, memory pool*>. The update could be triggered by whether a persistent modifier of a transaction coming in. Below we provide some implemented tests for the node view holder.

- *A totally valid persistent modifier should successfully update the minimal state and the history.*
  We recall that a totally valid modifier is a persistent modifier which is valid for both the history and the minimal state, so it is applicable to both of them. In this test we are sending a random totally valid persistent modifier to the node view holder component and then we are checking that history contains it and the version of the minimal state is equal to the modifier's identifier.

- *Forking tests.*

  At the moment we have two tests for forking. In both of them we apply random number of totally valid modifiers in the first place and remember last block which we call the common block. One test then is applying two totally valid modifiers which have the common block as parent. The test checks that after the application the history contains whether one of these modifiers, and version of the minimal state is equals to identifier of whether one of the two modifiers. The logic behind such a check is that we do not know whether an implementation of a node view holder will make switching from one prefix (of length one) to another (of the same length), but anyway the general property should hold. Another test first generates a sequence of totally valid persistent modifiers of length 2, applies it to the common block, then the test generates a sequence of totally valid persistent modifiers of length 4 starting from the common block, and applies the longer sequence. The test checks that switching takes place, so minimal state version equals to the identifier of the last block in the longer sequence, and the history contains the identifier in the current modifiers which do not have ancestors (for a blockchain, there is one such a modifier, for a block tree, there could be more than one modifiers returned). With the help of the forking tests we have found few errors in the Twinscoin implementation.

IV. Conclusion

In this paper we propose to improve quality of blockchain protocol implementations via exhaustive property-based testing. For generic abstract modular Scorex framework, we have implemented a suite of property-based tests. The suite consists of 59 tests checking different properties of a blockchain system. To run the suite against a concrete blockchain protocol client, developers of the client need to provide generators for random objects used by the protocol. The suite is checking properties against the implementation by using random samples. We used Twinscoin implementation provided with Scorex as an example of a concrete blockchain using our testing kit. In the paper we provide many examples of the tests.

Acknowledgments

Authors would like to thank Bruno Woltzenlogel Paleo for his valuable comments and corrections.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014, available at https://ethereum.github.io/yellowpaper/paper.pdf.
[2] P. Todd, "The difficulty of writing consensus critical code: the sighash_single bug," available at https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2014-November/006878.html.
[3] B. Wiki, "Value overflow incident," available at https://en.bitcoin.it/wiki/Value_overflow_incident.
[4] D. Ron, "Property testing," *COMBINATORIAL OPTIMIZATION-DORDRECHT-*, vol. 9, no. 2, pp. 597–643, 2001.
[5] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.
[6] T. Jung, "Quickcheck for java," 2015.
[7] B. Venners, "Scalatest," 2009.
[8] R. Nilsson, "Scalacheck: the definitive guide," 2014.
[9] G. J. Holzmann and D. Peled, "An improvement in formal verification," in *Formal Description Techniques VII.* Springer, 1995, pp. 197–211.
[10] M. H. Zaki, S. Tahar, and G. Bois, "Formal verification of analog and mixed signal designs: A survey," *Microelectronics journal*, vol. 39, no. 12, pp. 1395–1404, 2008.
[11] L. Goodman, "Tezos: A self-amending crypto-ledger," available at https://www.tezos.com/static/papers/position_paper.pdf.
[12] H. website, "Hyperledger sawtooth," available at https://hyperledger.org/projects/sawtooth.
[13] ——, "Hyperledger fabric," available at https://hyperledger.org/projects/fabric.
[14] B. Group, "Exonum - a framework for blockchain solutions," available at https://exonum.com/.
[15] IOHK, "Scorex 2.0 code repository," available at https://github.com/ScorexFoundation/Scorex.
[16] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "Spectre: A fast and scalable cryptocurrency protocol," Cryptology ePrint Archive, Report 2016/1159, 2016, http://eprint.iacr.org/2016/1159.
[17] A. Chepurnoy, T. Duong, L. Fan, and H.-S. Zhou, "Twinscoin: A cryptocurrency via proof-of-work and proof-of-stake," Cryptology ePrint Archive, Report 2017/232, 2017, http://eprint.iacr.org/2017/232.
[18] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.
[19] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE transactions on software engineering*, no. 3, pp. 178–187, 1978.
[20] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[21] W.-T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, “End-to-end integration testing design,” in *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*. IEEE, 2001, pp. 166–171.

[22] N. Tillmann, J. de Halleux, and T. Xie, “Parameterized unit testing: Theory and practice,” in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 2. IEEE, 2010, pp. 483–484.

[23] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver, “Stressing out: Bitcoin "stress testing",” in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 3–18.

[24] J. Garay, A. Kiayias, and N. Leonardos, “The bitcoin backbone protocol: Analysis and applications,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 281–310.

[25] “Spv mining.” [Online]. Available: https://bitcoin.org/en/alert/2015-07-04-spv-mining