TON VM Debugger

Description of use cases and restrictions.

User Interface (UI), off-chain services and database tests are similar to how traditional web apps are normally tested. In this order of ideas, we will now focus exclusively on the novel component of blockchain applications - that is, the blockchain and smart contracts functionalities. The QA principles and methodologies will remain the same: building requirements-based tests, automating and executing them, and carrying out risk assessments on the software for the development team.

However, when dealing with tests on a blockchain, there are a number of specific characteristics of this type of system that you have to consider before actually starting the tests. On the one hand, transactions on a blockchain are irreversible and may consume tokens or cryptocurrency from the account that originates them. On the other hand, performance-related tests on the blockchain (such as network scalability and volatility, functional integrity, data synchronization between nodes, etc.) will probably require that the network topology can be manipulated through the elimination or inclusion of new nodes in the blockchain, so it will be necessary to have another dedicated blockchain for this purpose. Another factor to keep in mind is that network nodes are often heterogeneous. As such, depending on the type of blockchain we are dealing with, we may have to evaluate them individually to obtain metrics to allow us to set expectations on the overall performance of the blockchain network. With regards to smart contracts, it is important to make sure that they work correctly as early as possible when they are developed, since once they are deployed they cannot be modified. If something needs to be changed in a smart contract there will be no choice but to destroy it and replace it with a new one. In addition, the best way to verify a smart contract’s behavior in different situations and without taking any risk is to test it while it’s being developed

Test levels

According to the practical test pyramid standard, we will build our first line of defense against bugs from the base of the pyramid with unit tests. Given the relative difficulty of smart contract debugging at runtime compared to other pieces of code, it is likely that many errors in our code will not become apparent until the unit tests are run.

A good example of this is the fact that to print messages that help us follow the execution of our contract to the console in Solidity - the most popular smart contract programming language - we

must resort to emitting events since this language does not support native logging functions. Likewise, in order to execute a contract, we will always need a node or part of a blockchain network to deploy it in. It is a good idea to use tools such as the Truffle framework, which have an integrated blockchain node to deploy and test the contracts, which greatly speeds up code executions during development. Some basic tests that you should definitely run on a smart contract are verifying overflows in data types (overflows, underflows); verifying that the return values of the functions are within the expected range; always testing limit values on functions, and ensuring that the return values come in the correct format and that the function parameters reject invalid values. Once the unit tests have been directly performed on the smart contract, integration tests will be the next test level. With this test we will verify that the unit component outputs cause the expected effects on the component that consumes them. Integration at this level can be both between smart contracts and between smart contracts and other components of the application that also use smart contracts. As with unit tests, integration tests are also run on code and we can automate its execution.

Finally, at the top of the pyramid we find system tests, or end-to-end tests. While in the previous test levels the responsibility of testing generally rests with the developers themselves, at this level, the role of the tester will probably emerge to take care of the tests.In system tests, testers need to address applications interactions to and from the blockchain ecosystem. For example, when a transaction is sent to an API, that transaction must be validated against a specific set of rules to generate an update order on the blockchain, after which the API will receive confirmation that the blockchain has been updated. Here, the testers must validate that the API requests and responses have the correct format and are handled properly.Regarding the information found in the blockchain, it contains a list of cryptographically secure records known as blocks. The block structure may vary from one blockchain to another, but in general they contain records such as: previous block hash, block header, current block hash, trie transactions, nonce, gas limit, gas used, logs, difficulty, block number. Blockchain results can be checked by analyzing real information on the network or by sending requests to the test blockchain indexing site.Among block records, it is of particular interest for our tests to inspect the transaction tree, to validate the status of the transaction generated by the application. In the transaction detail we can find data such as its status, timestamp, source address, destination address, transferred value, number of confirmations, gas used (in the case of Ethereum), commission paid.Some useful tools while running these tests will be: a blockchain network indexer (etherscan.io or similar), an event tracking and test case management tool, a local blockchain node, API testing tools (Postman, SoapUI, JMeter), database testing tools, and coding and encryption software, among others.

Performance tests

To finish the blockchain infrastructure analysis, we may also want to evaluate performance. Here it is a good idea to run performance tests on the blockchain both from the perspective of the application user and the required responses from smart contracts and other integrated systems. These performance tests will aim to identify bottlenecks, setting metrics to optimize the system and decide whether an application is ready for production.Something to keep in mind with these tests is the need to anticipate variations in the results, since the response speed may vary depending on the size of the P2P network which the blockchain is mounted on, as well as the transaction volume at a given time.

Smart contract testing

Functional tests for these contracts will evaluate use case scenarios as well as related business processes, such as the behavior of smart contracts. It is noteworthy that smart contracts are relatively difficult to verify because their execution allows them to interact with other smart contracts and off-chain services, and they can be repeatedly invoked for transactions from a large number of users. One way to handle smart contracts testing is the following:

- Verify contract integrity. That is, that the syntactic implementation follows best practice guidelines and that it behaves fairly, adhering to the agreed upon high-level business logic for interaction.
- Validate smart contracts' methods. Similar to API tests, we perform method validations, border value analysis, decision tables, TDD and BDD techniques, etc.
- Validate smart contracts compilation and deployment. Contracts are deployed in a similar way to a blockchain update for a cryptocurrency transaction and we need to validate that the block with such transaction has been created.
- Validate smart contracts processing. Monitor the execution of the smart contract code and ensure that it complies with the pre-established terms, regulatory tests, performance tests and business functionalities of the systems that use it.

Controlling smart contracts

There are a number of risks to control in smart contracts, such as:

- Sub-optimized code. For example we find risk with dead code, opaque predicates, expensive operations, constant results for different inputs, repeated counts.
- Programming errors. These can include calls to functions that do not exist, sending ether to orphaned addresses, unexpected execution when the transaction needs to consume more gas than it was assigned, irregularities in exceptions handling, typing errors, exposure of fields that should be private.

- State machine coding errors. Logical errors when coding the state transitions, such as omitting certain transitions or forgetting to verify the current state.
- Skip cryptography use. If input data is sent in plain text, a malicious user could wait to see the content of another related transaction before sending their own to make a change that is convenient for them.
- Misaligned incentives. It may be necessary to put page mechanisms in place to prevent users from failing to send an expected transaction due to a lack of incentives.
- Call-back overflow. While smart contracts can interact with each other by chaining calls, the general call stack is limited to a fixed size and if the call depth exceeds the limit an instruction could be skipped and cause unexpected results.

Security in smart contracts

From a more security-related approach, smart contracts present a combination of unique challenges because, since they can handle money in the form of cryptocurrency, they may well be an attractive target for potential attackers. Along the same line, as we mentioned earlier, unlike other pieces of software, once installed, smart contracts cannot be corrected directly. In addition to this, because they are developed with lesser tested programming languages and execution environments, these programs can be expected to be more susceptible to being harmed by attacks that are difficult to anticipate. On top of all this, network participants (miners or validators) also have the ability to manipulate parameters that influence smart contracts' execution through actions such as deciding which transactions to accept, how to arrange transactions, modifying the time stamp in the block, etc. Therefore, testing smart contracts that rely on such conditions will require keeping in mind these subtle semantics in order to explicitly protect the contracts against them.In order to face all these security challenges, we must perform different systematic analysis to detect and characterize vulnerabilities. On the one hand, we need to conduct static and dynamic code analyses to detect pattern violations and to verify semantic compliance; and on the other hand, we need to perform sequence analyses of invocations in order to detect vulnerabilities derived from the execution of multiple invocations to the same smart contract during its lifecycle.A tool that can help us with these analyses is Securify, a fully automated smart contract security analyzer that checks contracts' behavior by classifying them as safe or unsafe in relation to a particular characteristic. To do this, first it performs a symbolic analysis of the contract dependencies in order to extract semantic information from the code, and then registers conditions following vulnerability patterns to finally determine whether a security property is met or not. Other examples of such tools include SmartCheck and MythX.As for the invocation sequence analysis, a useful tool for this purpose is Maian. Unlike the tools we already mentioned such as Oyente or Securify that apply static and dynamic analyses to automatically find errors in contract executions, Maian focuses on detecting vulnerabilities through long

invocation sequences in contracts. These vulnerabilities make up what is referred to as an execution trace. Maian then classifies the vulnerabilities in the execution trace according to the three types of known problems that they can cause: contracts that block funds indefinitely, contracts that lose funds indiscriminately towards random users, and contracts that can be terminated by any user.To do this, Maian uses interprocedural symbolic analysis and a specific validator directly on the bytecode of smart contracts (not requiring access to the source code), which allows it to accurately distinguish and specify the execution trace vulnerabilities and identify problems that it will divide according to the aforementioned known problems into the categories of greedy, prodigal and suicidal, respectively. This characterization allows us to confirm if the errors found are true positive or not, by manually executing the contracts. At this point, we know that beyond rigor tests on the application's graphical interface, off-chain services and the relational database (which have not been discussed in depth in this article), several other tests have been performed:● Unit tests, during the development of smart contracts.

- Tests on the blockchain ecosystem, both in its internal and its integration with other subsystems.
- Verification of smart contracts, not only its functionality but also the syntactic correction and code semantics using automated tools that validate compliance with good practice patterns and verify known vulnerabilities

Techical & Functional Specifications.

TON Project System comes with a simple and intuitive layout. The UI is divided into several pages, switchable through the navbar buttons.

- Project Editor is the main interface to display project and provide a editor to code, build and deploy your smart contracts.
- Contract Inspector is a convenient tool to debug smart contracts. It allows you to easily execute contract actions and visualize the table data.
- Account Viewer is a page to view account information and perform account related operations.
- Network Manager can help you switching between local network, different testnets, and FREETON Mainnet, as well as showing the information of the selected network.

The current landscape of Smart Contract development is one that does not truly enable developers to debug transactions comprehensibly during execution time, and while some debuggers like Remix or EVM.Debugger and a GUI implementation of it exist, and all of them do their job well, there is no debugging tool available that features:

- A comprehensible GUI,
- the possibility to view transactions during execution time, before being mined,
- loading contracts without ,
- supports external calls and
- platform independence*

Contract Inspector aims to resolve these issues. Using a GUI, users can load contracts and interact with them by sending transactions, either by using a predefined ABI or raw, while following the resulting changes step-by-step.

Adding WebAssembly as one of variant to the TON SDK allows software written in many languages to run securely on a blockchain. WASM serves as an intermediate language that compiles the developer's language of choice into a portable virtual machine. This means that you can have a simple, secure and fast virtual machine set up to sandbox or partition your application's actions for better testing, security, performance and speed.

A suite of development tools for smart contracts.

The total state of TVM consists of the following components:

- Stack (cf. 1.1) — Contains zero or more values (cf. 1.1.1), each be-

  longing to one of value types listed in 1.1.3.

- Control registers c0–c15 — Contain some specific values as described

  in 1.3.2. (Only seven control registers are used in the current version.)

- Current continuation cc — Contains the current continuation (i.e., the code that would be normally executed after the current primitive is completed). This component is similar to the instruction pointer reg- ister (ip) in other architectures.
- Current codepage cp — A special signed 16-bit integer value that selects the way the next TVM opcode will be decoded. For example, future versions of TVM might use different codepages to add new opcodes while preserving backward compatibility.
- Gas limits gas — Contains four signed 64-bit integers: the current gas limit g_l, the maximal gas limit g_m, the remaining gas g_r, and the gas creditg_c. Always$0 \leq g_l \leq g_m, g_c \geq 0$,and$g_r \leq g_l + g_c$;$g_c$ isusually initialized by zero, g_r is initialized by $g_l + g_c$ and gradually decreases as the TVM runs. When g_r becomes negative or if the final value of g_r is less than g_c, an out of gas exception is triggered.

That's why we should have orientation on this SCCCG structure when creating debugger tool.

What it should be

- Built-in smart contract compilation, linking, deployment and binary management.
- Configurable build pipeline with support for custom build processes.
- Scriptable deployment & migrations framework.
- Network management for deploying to many public & private networks.
- Instant rebuilding of assets during development.

The Contract Page provides the necessary tools to inspect and debug smart contracts. In order to view multiple contracts at the same time,

The Contract Inspector should have two parts:

1. a panel to execute actions on the left, and
2. a panel to query table data on the right.

Actions

Actions are shown on the right. You can switch the action you want to call through the dropdown menu.

Form for Primitives

A form for primitives will be generated to make it easier to enter parameters.

The input of the action contains many types will process the input parameters according to the type:

- Stack manipulation primitives
- Tuple, List,and Null primitives
- Constant, orliteral primitives .
- Arithmetic primitives.
- Comparison primitives
- Cell primitives.
- Continuation and controlflow primitives
- Exception generating and handling primitives
- Dictionary manipulation primitives
- Application-specific primitives
- Debug primitives
- Codepage primitives

Authorization

You can change the actor and permission used to sign the transaction.

Components

Account

An Account is basically an Address in the FreeTon Blockchain

Balance

Every Account comes with a balance. Accounts that have never been interacted with (should) have a balance of 0. There are several ways to increase the balance of an account, the simplest being just sending Crystal to it from another account.

Contract

A contract is an account under which additional code is stored. Transactions that are sent to a contract will trigger the execution of this code, and depending on the data that is sent along a transaction, different parts of the code gets processed.

Storage

Every contract possesses a storage under which data can be persisted. The operation of storing data is a lot more expensive than other operations, mostly because in a real world block chain

application the storage needs to be mirrored on every node, and thus comes with permanent costs for each miner.

Transactions and Messages

A transaction isn't just a transaction in the financial sense, but in logical way, which means that either the transaction is executed and the resulting state is stored entirely, or the changes are dismissed. Each transaction comes with the following attributes:

Origin

The original account that has triggered this transaction.

Sender

The sender of a transaction. This value can differ from origin e.g. when you call a contract function that triggers another transaction.

To

The target address of a transaction.

Value

The amount of wei that a transaction holds.

Data

The data field needs to be specified e.g. when you want to call a contract's function.

Call depth

This property shows the depth of the current call. Whenever a contract makes further calls internally, this value gets incremented by one.

Gas Limit

Specifies how much gas to send alongside a transaction.

Opcodes Table

Shows the contract's entire code, alongside the mnemonic name and gas cost estimate inside a table.

Stack Table

Shows the stack of the current execution context. Since the TVM is a stack machine, and most opcodes interact with the stack in a certain way, the relevant stuff is most likely to happen here.

Memory Table

Shows the memory of the current execution context. The memory is a separate area in which a contract can read and write data during a transaction's execution data, however the memory's content is not persisted.

Storage Table

The storage of the currently shown contract.

Some of the key in the runtime include:

- **Balances** — Support for accounts, balances, and transfers.
- **TVM** — Full Rust-based TVM implementation based.Provides the state transition logic for smart contracts.
- **RPC-TON** — Web3 RPC implementation in Substrate.
- **Council** — Includes governance mechanics around the council and proposals.
- **Executive** — Orchestration layer that dispatches calls to other runtime modules.
- **Indices** — Support for user-friendly shortnames for account addresses.
- **System** — Provides low-level types, storage, and blockchain functions.

Technology

Rust Programming Language

Rust is a good language for implementing a blockchain, as it is highly performant like C and C++, but has built-in memory safety features that are enforced at compile time, which prevents many common bugs and security issues that can arise from C and C++ implementations.

Framework

Framework provides a rich set of tools for creating blockchains, including a runtime execution environment that enables a generic state transition function, and a pluggable set of modules that provide implementations of various blockchain subsystems.

Blockchain Runtime

Smart contracts can be implemented using Solidity, and any other language which can compile smart contracts to TVM.

A high-level interaction flow is shown above. A Web3 RPC call from a DApp or existing developer tool, such as Truffle. The node will have both Web3 RPCs and RPCs available, which. These RPC calls are handled by associated Framework runtime functions. The Framework runtime checks signatures and handles any extrinsics. Smart contract calls are ultimately passed to the TVM to execute the state transitions.

Roadmap

- Provide implementation for RPC & Web3(optional)
- Provide web interface/desktop app for develop smart contracts
- Provide interface for automate testing and security testing

The final implementation will depend on all architecture of project

Telegram: @ducktalesblock
E-mail: info@itty-bippy.space
**FREETON Address:
0:c0efbaaa82ea20862cc7b49fdd57288275eae56ff92832c5c948a37943888691