Security Issues of Smart Contracts in Ethereum Platforms

Tomas Krupa, Michal Ries, Ivan Kotuliak, Kristi´an Košťál, Rastislav Bencel
Slovak University of Technology in Bratislava
Bratislava, Slovakia
{xkrupat, michal.ries, ivan.kotuliak, kristian.kostal, rastislav.bencel}@stuba.sk

***Abstract*—Smart contracts belong to the most rapidly growing aspect of the world of cryptocurrencies. This phenomenon attracts great attention from researchers but also the business community, and the development brings daily a lot of novel applications. Smart contracts allow running contract code transparently for all parties, without the need for a centralized authority. One of the smart contract applications is the issuance of digital assets, so-called tokens, serving as fundraising fuel for Initial Coin Offerings. ICOs bring a new easy, and bureaucracy-less way for startups to raise considerable funds from crowds with incredible speed. However, the technology hides a dark side in the form of speculative scams, hardly distinguishable from genuine fundraising activities. Evaluation audit of ICOs associated with the underlying security of smart contracts is a complex issue requiring many efforts.**

This paper focuses on one of the most popular blockchain frameworks, Ethereum, a prominent ICO and smart contract platform, and its dominant programming language, Solidity.

I. Introduction

From a technical point of view, the smart contract is just a computer code as any else. Humans make it, and it is error-prone, either unintentionally or intentionally. The critical difference between a blockchain-based environment and any other software is how the code is deployed and executed. Intervention to standard runtime-executed software, whether it is a simple update, upgrade, or bugfix, means simply changing existing code to a new one on the fly or accompanied with a system reboot. Every kind of code maintenance is feasible, whether it concerns thousands of computer devices or a giant cloud system - the key factors allowing this are ownership and competence. On the other hand, these two factors are missing in the blockchain-based ecosystem. As the environment powered by the Ethereum Virtual Machine (EVM) is a giant decentralized unstoppable machine, where are no downtimes or regressions possible. The deployed code is immutable and executed permanently without interference or censorship. Once the code is deployed on the blockchain, it will be executed everywhere until it provides resources (i.e., Gas). In a case the code is malformed, contains a vulnerability, or just mistaken steps, there is no legitimate way to terminate or remove it unless it provides such a functionality itself. We can only deploy a new smart contract and broadcast the information that the old one is deprecated, and everyone should use the new address [1]. However, the Byzantium upgrade (https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement/) included a `revert` opcode that allows a contract to halt execution.

There is also a not-so-familiar option, maybe the ”non-legit” one, i.e., a fork of the blockchain [2]. The fork means that we change something fundamental in the blockchain, which creates a new version, which is not continuous to the previous blockchain implementation. Either revert the blockchain in time to the state before the fraud smart contract was deployed or update EVM's source code to eliminate the exploit. Blockchain forks are the most severe interventions [3] to the cryptocurrency ecosystem, accompanied by the separation of participants, capital devaluations, and asset pseudo-duplications. When a cryptocurrency blockchain is forked, the pre-fork coin is acceptable in both forks. Thus, twice-time exchangeable for legal tender, even base capitalization has not changed, consequently leaving some mutual interconnection between the two cryptocurrencies' balances.

In the current state of the art [4], all the cryptocurrencies are in continuous development, and none is fully deployed as a global legal tender. The forks can be seen multiple times in a year, but they are more crucial as the currency gets larger. When a cryptocurrency is deployed as a widespread legal entity in a hypothetical future, maybe there would be no forks acceptable. Analog could be a complete exchange of some fiat currency's whole physical value, e.g., USD.

These characteristics and consequences are reasons to develop and deploy a powerful mechanism to a security audit of any smart contract before being deployed into the network [5]. The purpose of such a mechanism is not to restrict or filter any smart contract before deployment. It is principally not possible because of the blockchain environment's character, but it should be a powerful instrument avoiding people from participating with an unreliable smart contract and losing their money. On the other hand, it could enhance such a smart contract and its developers' credibility and trust as it would pass the audit.

The paper is organized as follows: Section II talks about smart contracts in Ethereum and its programming language, Solidity. It also mentions drawbacks and some security issues of these smart contracts. We analyze smart contract design

ISSN 2305-7254

patterns in Section III and propose recommendations to avoid the most well-known attacks or problems. Section IV discusses smart contract code optimization with an emphasis on gas price. Furthermore, Sections V and VI are going a little bit deeper in the domain of smart contracts, mentioning tokens and Initial Coin Offerings, respectively. Next, Section VII summarizes functional validation of Initial Coin Offerings, and in Section VIII, we are describing our favorite tools to mitigate the Ethereum pitfalls. We summarize the paper in Section IX.

II. Smart contracts in Ethereum

The underlying layer of smart contract code is a low-level, stack-based bytecode language, executed by Turing-complete Ethereum Virtual Machine [6]. It contains a predefined set of instructions sequentially executed on each EVM node in the network independently, reaching mutual consensus via distributed computing mechanisms. To guarantee the honest execution of the code in the global network, each instruction (opcode) is weighted by a cost, measured in units of Gas [7]. It is an indefinite component of a transaction's financial asset. The more computational power the code needs, the more Gas resources it consumes. This financial association with code execution provides a fair sharing of computational power and secures the network against DoS attacks carried by time-consuming computations [8]. Simultaneously, the Gas is a transaction fee, motivating EVM operators to participate and execute the contracts, so the consumed Gas is accounted to their balances.

Smart contract code is submitted to the network the same way as a direct coin transaction. The user passes it to a node right away or via a web-service. The node buffers the transactions and broadcasts to all its neighbor nodes, so they are gradually widespread to the whole P2P (peer-to-peer) network. When pulled from the buffer, the smart contract is deployed to the blockchain as a transaction in (mined) block's transaction trie [9]. It becomes an "account" obtaining a wallet address as a unique identifier. Afterward, it is being executed autonomously in the block validation phase. Participants can interact with the contract by sending contract-invoking transactions to that address. Execution of the contract may result in various actions, e.g., modification of the contract's state, proceeding input data, triggering different contracts, or transferring assets to another wallet. After the network executes the contract, its result state is reflected in the state database, program memory in storage trie, and transaction output in the next block's receipt data.

A. Drawbacks of smart contract technology

From a technical perspective, a blockchain ecosystem is not a universal program environment suitable for every use-case. Its inherited characteristics as decentralization, transparency, immutability, or execution redundancy are not a must of most software, as well as stack-based architecture is not practical for the vast majority of programming constructions. After all, computation-intensive tasks are not feasible to run on public blockchain because of computation fees and lack of performance. Also, the architecture of fully redundant nodes, which are not sharded, does not favor storing a large amount of data, which leads to the enormous size of the distributed ledger.

These constraints predestine smart contracts for applications that benefit from being distributed and publicly verifiable and enforceable, e.g., money operations, elections, insurance, cadastre records, and various digital agreements [10].

B. Ethereum smart contracts programming language: Solidity

Smart contracts are written in high-level Turing-complete programming language before being compiled to EVM byte-code. The most popular and widespread contract-oriented language is Solidity. It is a statically typed language with Javascript- and C-like imperative syntax and the support of user-defined types, inheritance, and polymorphism [11]. Moreover, it provides a concept of libraries, i.e., contracts, as a reusable code that can be called from different contracts. In brief look into code syntax, Solidity contains special variables and functions (e.g. `block`, `msg`, `tx`, `gasleft()`) that are still present in the global namespace and are mainly used to provide general information about the blockchain or the invoking transaction.

Other features are function modifiers, which are intended to amend the semantics of functions in a declarative way. These inheritable properties of contracts can change a function's behavior and modify the flow of contract execution. The new function consists of modifier's body where keyword "_" is replaced by the body of the called function. This condition-oriented programming (COP) approach removes a need for conditional paths in function bodies.

Another Solidity features are events, seen as signals dispatched by smart contracts. They are based on convenient usage of the blockchain's logging facilities, which can be used to trigger Javascript callbacks in the listening interface, e.g., Decentralized Application (dApp). Decentralized application consists of web-hosted JS client hooked to blockchain-stored (smart contract-based) backend [12]. The concept gives a tangible form to a smart contract environment. Decentralized runtime and open source are guarantees of trustworthiness. The logs themselves are invisible for contracts but visible to the audience.

```
pragma solidity ^0.4.0;
contract Ballot {

    struct Voter {
        uint weight;
        bool voted;
        uint8 vote;
    }
    struct Proposal {
        uint voteCount;
    }

    address chairperson;
    mapping(address => Voter) voters;
    Proposal[] proposals;

    function Ballot(uint8 _numProposals) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
        proposals.length = _numProposals;
    }
```

```
    function giveRightToVote(address toVoter) public {
        if (msg.sender != chairperson || voters[toVoter].voted)
            return;
        voters[toVoter].weight = 1;
    }

    function vote(uint8 toProposal) public {
        Voter storage sender = voters[msg.sender];
        if (sender.voted || toProposal >= proposals.length)
            return;
        sender.voted = true;
        sender.vote = toProposal;
        proposals[toProposal].voteCount += sender.weight;
    }

    function winningProposal() public constant returns (uint8
          _winningProposal) {
        uint256 winningVoteCount = 0;
        for (uint8 prop = 0; prop < proposals.length; prop++)
            if (proposals[prop].voteCount > winningVoteCount) {
                winningVoteCount = proposals[prop].voteCount;
                _winningProposal = prop;
            }
    }
}
```

Listing 1. A simple ballot contract where eligible users can vote for desired proposal, source: sample code from Remix IDE

With the given brief description, we can analyze the smart contract sample shown in Listing 1. First of all, the example is a simple ballot allowing an eligible person to vote once. The proposal with the highest votes wins. The author of the ballot can select the eligible addresses which can vote for a proposal.

On the first line, the compiler version is set to ensure compatibility. We do not recommend to use ^ notation, visible in the sample, to avoid untested behavior on newer compilers. It would be better to specify the exact version of the compiler.

The contract itself is then defined, followed by its state variables, i.e., struct, address data type, and address mapping. Then follow three contract functions. The constructor `Ballot` is a special function callable only once, during the contract's creation. It stores the number of proposals and address of the creator, i.e., the ballot owner, for a future adding of voting rights. The other functions serve for interaction and can be called by users or contracts. The function `giveRightToVote` checks that only the owner can be callee and voter has not yet voted, and gives him a vote. The next function `vote` checks if the voter has not yet voted and if the proposal number is not outside boundaries and consequently assigns vote to the proposal. Keyword `storage` creates reference to sender instead of memory copy, so the changes of `sender` object are provided to the `voter`'s item. Finally, the function `winningProposal` iterates over all the proposals and return a winning proposal.

To sum this up, the sample code explains the fundamental concepts of smart contracts developed in Solidity. It depicts one of the most critical capabilities of a smart contract: the power to manipulate a worldwide verifiable and universally consistent contract state (i.e., balances) [13].

C. Smart contract issues

We distinguish four fundamental categories of Solidity issues affecting the development of smart contracts (sorted descending by severity) [14]:

- Security issues open doors to harmful actions by hackers.
- Functional issues are not exploitable by malicious users, rather cause unintended functionality.
- Operational issues result in runtime problems, such as poor performance and high Gas consumption.
- Development issues represent mainly bad-habits and antipatterns leading to unreadable code, which is impossible to improve.

III. Smart contract design patterns

The cryptocurrency ecosystem is in hackers' focus due to its anonymous environment and many financial assets being at stake. Nearly 99.9% of contracts have issues, and ˜63% of them have a critical vulnerability [15].

There exist several smart contract design patterns in the context of security execution. They address typical security issues mentioned above and further vulnerabilities which can be safely mitigated by applying such a security pattern. Following vulnerabilities are ordered descending by severity.

A. Re-entrancy attack

Normally, when a contract calls another contract (e.g., with a coin transfer), it hands all the control. The called contract then can call the former contract and change its state variables, causing, e.g., multiplying the transfer. We recommend applying the "Checks-Effects-Interaction" pattern (https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html), which defines the exact order of actions:

1) check all preconditions,
2) perform all changes to the own state,
3) call the other contracts.

Ensuring the secure order with contract interaction as the last avoids such vulnerabilities.

For even more security, we recommend incorporating Mutex lock over the smart contract's invoker. Sender's fallback function may implement a call to the smart contract's delicate function (e.g., fund transfer), allowing a recursive call to it and thus a re-entrancy attack. Incorporating a mutex variable into functions restricts concurrent executions from external contracts and avoids re-entrancy.

B. All-Gas-forwarding transfer

Solidity call `call.value(x)()` transfers `x` ethers and forwards all Gas to the addressee, leaving a potential vulnerability for an attacker, especially for re-entrancy. There exist another two functions to send ethers between addresses, the first, which is older, `<address>.send(amount)`, and the newer one, `transfer(amount)`. When determining the characteristics, there are two dimensions to consider: the volume of forwarded Gas and the propagation of exceptions. A stipend of 2300 gas is forwarded for each `send` and `transfer`, which is only enough to record an event at the receiving contract. If the receiver needs a greater quantity of Gas, `call.value` must be used as it forwards all remaining Gas, unless otherwise stated with the aid of the `.gas()` parameter. With regards to the exceptions, in the case of an error, `send` and `call.value`

are identical, either return false and do not bubble exceptions. However, the `transfer()` method propagates each exception thrown at the receiving address to the transmitting contract. Although both methods, `transfer` and `send` are considered secure against re-entrancy since only 2300 gas is transferred, we recommend that the `transfer` should be the go-to method for transferring ether in most cases. This is because it reverts immediately in case of any bugs. The `send` function can be seen as the transfer's low-level counterpart. It should be used in situations where the fault must be dealt with in the contract without reversing all state changes. The low-level `call.value` approach should be used only as a last resort, as it violates Solidity's type-safety.

C. Manipulation via external call

Using external calls without a return value check may be abused by an attacker via a counterfeit function. Even a seemingly simple send call can be foiled by a fallback function that intentionally runs out of Gas. We recommend always wrap external calls into if condition to check unexpected failure. Moreover, we recommend using `require` and `assert` as much as possible throughout the code. The engine considers `require` statements as assumptions and tries to prove that `assert` statements' conditions are always true. Also, we prefer using `transfer` instead of `send` which misses `throw` semantic.

D. DoS via external call

External call returning variable used inside if condition or loop can be used by an attacker to Denial-of-Service attack the contract, the most often seen vulnerability [15]. The callee can permanently fail using `throw` or `revert` to keep the caller from accomplishing execution. That is why we always use it outside these scopes.

E. Man-in-the-middle with tx.origin

An attacker can misuse the presence of `tx.origin` in the code by chaining calls to the contract allowing to fake identity. If our wallet had selected `msg.sender` for authorization, instead of the owner's address, it would get the attack wallet's address. However, by selecting `tx.origin`, the original address that began the transaction, which is still the owner's address, is obtained. The assault wallet steals all our funds immediately. `tx.origin` should always be replaced with `msg.sender` which return true function invoker, not the chain originator.

F. Forced branching manipulation

Using contract balance in if conditions with not covering all possible states may let attacker manipulate contract states by, e.g., forcibly sending ethers via `selfdestruct()` or by mining. Avoid checking with strict (in)equality and rather use <=, >= conditions.

G. Lost control over execution

Even well-working smart contracts can contain hidden bugs revealed by a harmful attack. As they are executed on the network autonomously, there is no option to terminate them in case of a bug. We recommend incorporating Emergency Stop, i.e., explicit halt functionality callable by the contract owner. Such a triggerable stop can disable a vulnerable function (or entire smart contract) that uses it.

H. Uncatchable execution

Massive simultaneous execution of sensitive tasks (e.g., withdrawals) can bring damage to the smart contract. Incorporation of a Speed Bump, i.e., time delayer (or periodic switch), into the delicate functions can help use an emergency stop to counteract a fraudulent activity in time or prevent drainage of funds before it emerges.

I. Too wealthy contract

With the growing balance of the contract, the risk is rising too. We recommend following the Balance Limit pattern to the upper limit of the contract's balance. It means incorporating a simple deposit rejector when the limit quota is reached. As it applies only to the payable function, it does not concern forcibly sent assets, e.g., from self destruct call.

J. Functional issues

Among them, we distinguish:

- *Locked money* problem, when the contract does not provide any withdrawal method even if it contains payable calls,
- *Unchecked math* when using raw mathematical operations instead of safe mathematical libraries,
- *Integer division* rounding the result down and possibly cutting ethers,
- *Unsafe type inference* when comparing different-size variable, e.g., in loops, causing overflow,
- *Timestamp dependency* what can be guessed by miner giving the ability to manipulate execution.

The introduced issues are checked via static analyzers [16], inspecting uncompiled source code semantic Abstract syntax tree (AST) tree and formal methods. See chapter Tools to mitigate the impact of Ethereum pitfalls for recommended tools used in our environments.

IV. Smart contract code optimization

As EVM's instructions execution is priced with Gas units, code effectiveness is a delicate issue in smart contract development. Transaction sender pays upfront an estimated amount of Gas the execution will consume and gets a residual refund in case of accomplishment. In case of execution abort, due to code exception or run out of Gas, the state is returned, but Gas is not in many cases. Overloading transaction with Gas is also not always a solution as EVM nodes may dismiss them to avoid time-intensive execution. Due to this, Gas estimation

must be done precisely, ideal with a well-proven tool as, e.g., ETH Gas Station (https://ethgasstation.info/).

The Table IV depicts approximate gas price of Solidity assembly calls. There are three columns in the table inspired by and took from Ethereum's Yellow Paper Appendix. The first column is the name of function, the second column is value in Gas and the last column is description of the function.

TABLE I. GAS PRICE OF EVM OPCODES[1]

ADD/SUB	3	Arithmetic operation
MUL/DIV	5	Arithmetic operation
ADDMOD/MULMOD	8	Arithmetic operation
AND/OR/XOR	3	Bitwise logic operation
LT/GT/SLT/SGT/EQ	3	Comparison operation
POP	2	Stack operation
PUSH/DUP/SWAP	3	Stack operation
MLOAD/MSTORE	3	Memory operation
JUMP	8	Unconditional jump
JUMPI	10	Conditional jump
SLOAD	200	Storage operation
SSTORE	5,000/20,000	Storage operation
BALANCE	400	Get balance of an account
CREATE	32,000	Create an account using CREATE
CALL	25,000	Create an account using CALL

[1] https://github.com/crytic/evm-opcodes

More than 90% of deployed contracts contain dead code or opaque predicates, and more than 70% use gas-costly loops [17], means the issue of under-optimized code is bothering almost all smart contracts. Therefore we recommend using minimum external calls to a third-party library, as well as minimizing the number of functions (and function arguments) and data redundancy in global storage. Avoid using costly loops like array iterations and use `bytes` instead of `byte[]`. Smart contract-oriented programming does not strictly follow OOP patterns and perfect code readability due to critical execution cost, similar to Real-time programming.

From a security point of view, we do not recommend using the auto-optimization features in the compilers. The optimizer operates on assembly (https://docs.soliditylang.org/en/latest/internals/optimiser.html) and such optimized code cannot be more inspected with a static analyzer, which demands text

source code. This brings in a potential security issue due to untested execution flow or bug from the optimizer itself.

Finally, we can once again look at the original function from the previous Listing 1 and demonstrate its optimized version on Listing 2 with lower gas consumption.

```
function winningProposal() public constant returns (uint8 _winningProposal)
    {
    uint256 winningVoteCount;
    uint8 prop = uint8(proposals.length);
    uint256 voteCount;
    for (; prop >= 0; --prop){
        voteCount = proposals[prop].voteCount;
        if (voteCount > winningVoteCount){
            winningVoteCount = voteCount;
            _winningProposal = prop;
        }
    }
}
```

Listing 2. Optimized function `winningProposal` from the Ballot contract

The optimized function has a smaller code size, uses fewer instructions in the loop, and uses cheaper instructions.

V. TOKENS

Solidity enabled the emerging of an asset extension called a token. Generally, the tokens are digital assets, a pseudo-currency running over a blockchain. In terms of Ethereum, the main difference between coins and tokens is that the coins are native digital assets, a challenging part of its blockchain. In contrast, the tokens are independent extensions, implementing their subset of rules, existing in their sub-ecosystem over such a blockchain.

In simple terms, tokens provide a way to deploy their independent cryptocurrency in an existing, well-run, stable, and widespread blockchain system. It relieves developers from developing their underlying layer and allows them to focus directly on the application layer of the currency. To be accepted by the EVM community, they must comply with a defined set of rules, i.e., ERC (Ethereum Request for Comment - an analogy of RFC mechanism) standard. These rules present a simple set of functions that the contract must implement. In return, contracts implementing the standard can be used via a common API.

We distinguish between fungible and non-fungible tokens. Fungibility means an ability to be replaced by another identical item. In terms of cryptocurrencies, a fungible token (e.g., ERC-20 or ERC-223) is entirely interchangeable with other identical tokens. An analogy can be fiat money; a €uro coin, perfectly exchangeable with another €uro coin. Fungibility is the essential feature of every cryptocurrency providing three fundamental properties: interchangeability, uniformity, and divisibility. On the other hand, non-fungible tokens (e.g., ERC-721 (http://erc721.org) standard) represent very opposite, except com-mon tradeability: non-interchangeability, uniqueness, and non-divisibility. They are not finding application in cryptocurren-cies, instead of in games and dedicated applications as cadastre records or car-sharings [18].

In ICO, we mainly see ERC-20 (https://github.com/ethereum/EIPs/issues/20), the most well-known token standard within the community, and most used one among issued tokens in the Ethereum blockchain. It defines oblig-atory interface with functions `totalSupply`, `balanceOf`, `allowance`, `transfer`, `approve`, `transferFrom` and events `Transfer` and `Approval`.

There was also a new proposal, ERC-223 (https://github.com/ethereum/EIPs/issues/223), or, by someone, called ERC-23, which is backward-compatible superset of ERC-20, fixing security issues of the predecessor, extensing the interface with `balanceOf` and `transfer`, and event `Transfer` overriding default ERC-20 interface. It was created to prevent accident sends of tokens to contracts and make token transactions behave like ether transactions. The problem here was the lack of possibility to handle incoming ERC20 transactions that were performed via transfer function of ERC20 token to smart contracts because smart contracts do

not recognize an incoming transaction; thus, the sent tokens to smart contract get inevitably lost.

VI. Initial Coin Offerings

The mentioned features made tokens finding application in crowdfunding, causing the well known ICO phenomenon [19]. Initial Coin Offering is a kind of public offering (similar to Initial Public Offerings (IPO) and Venture Capitals (VC)) of a new cryptocurrency in exchange for an existing one, intending to fund projects in the blockchain environment. It is being considered a state-of-the-art strategy of financing new ventures, as it minimizes transaction costs, democratizes financing, and disintermediate banks and bureaus [20]. Despite many efforts to ICO regulations, e.g., Swiss Finance Regulator and Singapore's Central Bank have already issued regulatory guidelines for ICOs [21], and even banned in several countries, the simplicity of sending funds via Ethereum transactions, and the effect of the rapid growth of investments even before the project gets to the production (because the tokens are tradeable immediately on dedicated exchanges) - caused the ICO phenomenon to explode [22].

By January 2020, the total funds raised by ICOs exceeded $26,5billion (https://icobench.com/report) and had overcome investments funneled through VC of high-tech initiatives, although their approximate effi-ciency is 35%. It is nearly 80% of active ICOs [19] being handled through contracts running on ERC-20.

Typical ICO crowdfunding consists of three phases:

A. Pre-ICO

Pre-ICO phase means token presale, often with discounted price. It runs on a separate smart contract to avoid mixing with the main phase and to ensure easy account reconciliation and audit. It is often in the preliminary stages of project development with no valuable parts yet delivered (usually just a white-paper).

B. Main-ICO

The main-ICO phase (or crowd sale) is the main stage of the token sale. The token should be tradable on dedicated exchanges by the end of the phase. The active main stage should guarantee the project's solvency, underlined by delivered valuable parts (yellow-paper, prototypes, large-scale marketing).

C. Post-ICO

The post-ICO phase basically defines a new set of rules for the token contract. Technically, it disables and enables particular functions, e.g., allows trading tokens. It defines events as well that take place as the main-ICO phase ends, like unsold token burn-up.

VII. ICO functional validation

Enterprises manage crowdfunding directly via token sales. The token smart contract means own Solidity implementation with reusing of available samples and patterns. There is a need to attract investors with a guarantee of reliability of their finances in stake. To build trustworthiness, the code must be well-tested from many aspects. Each phase of ICO means different use-cases and work-flow, requiring dedicated test-cases.

To ensure reliability, we recommend using multi-signature wallet [23] (so-called Multi-Sig). It is a smart contract that defines access rights to particular users (i.e., addresses), specific rules, and consensus conditions to operate. For instance, withdrawing a balance is not possible until all members vote. Using a Multi-Sig, ICO's balance cannot be laundered easily.

VIII. Tools to mitigate the impact of Ethereum pitfalls

Input for our testing stages is test specification extracted from design requirements and user stories, some sort of what it is expected to do and what it is designed to do. Our implementations are based-on OpenZeppelin (https://github.com/OpenZeppelin/) sample contracts which regarded as highly-secure. As we practice test-driven development (abbr. TDD), we write tests ahead. In non-functional unit tests, we tweak various inputs and outputs, threshold- and peak values. Functional integration tests are performed with interface mockings to plant managed behavior as well as simply eliminate possible error-proneness of the (non-mocked) interface. We use verbose logging to get the current stack state without entering the debugger. We use a debugger embedded in Remix IDE to step the execution line-by-line in case of a bug. After the automated tests pass, we perform manual testing on global testnet, primarily Ropsten network (https://ropsten.etherscan.io). We trigger the contract via other contracts as well as by submitting simple transactions. In this stage, there is the final form with no mocking present in the code. After Ropsten testing, we also perform tests on the Ethereum mainnet. We do so due to the environments' not fully-equivalent behavior, but only with a small number of assets.

A. Related tools

In our development process, we use JetBrain's IntelliJ IDEA(https://www.jetbrains.com/idea/) IDE with Solidity plugin (https://plugins.jetbrains.com/plugin/9475-intellij-solidity) for code editing, which provides excellent code editing tools. We compile code using Remix IDE, a browser-based tool providing comprehensive functionality. We use it also for static analysis of code as well as SmartCheck (https://tool.smartdec.net) and Securify (https://securify.chainsecurity.com) online analyzers. There is also an excellent tool to visualize a topology diagram of invocation relationship and to find potential logic risks [24].

We use Truffle-framework (https://truffleframework.com/) for automated unit and acceptance tests, with test cases described in JS. It runs Ganache, a personal blockchain framework, which we also use to inspect transactions (via its GUI). For deployment, we use Remix coupled with Metamask (https://metamask.io/) browser plugin, which submits the transactions to INFURA's (https://infura.io/) remote nodes.

IX. Conclusion

We provided an essential introduction ttEthereum platform and Solidity smart contract language with a comprehensive insight into code issues and ways to mitigate them. We provided a brief overview of Ethereum token standards and ICO mechanism and presented testing tools based on our development environment, which collaborated on some commercial products.

In this paper, we were focusing on the Ethereum platform and Solidity development environment. In future work, we plan to generalize our focus on multiple blockchain environments, and their smart contract languages, especially ecosystems with multi-chain and cross-chain [25] design and based on modern consensus, e.g., Proof-of-Stake [26].

Security is seemingly still the main problem of smart contract platform, preventing gaining of reputation and public widespread. We hope our advices will help to produce reliable smart contracts.

Acknowledgment

This research was supported by the Ministry of Education, Science, Research and Sport of the Slovak Republic, Incentives for Research and Development, Grant No.: 2018/14427:1-26C0. This publication was created thanks to supporting under the Operational Program Integrated infrastructure for the projects: Research in the field of blockchain technology with connection to online payment services, ITMS 313022U641, and Electronic methods for detecting unusual business operations in a business environment, ITMS 313022W057, both co-financed by the European Regional Development Fund. It was also partially supported by the grants APVV-15-0731, KEGA 011STU-4/2017, and VEGA 1/0836/16.

References

[1] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, "Smart contract security: A software lifecycle perspective," *IEEE Access*, vol. 7, pp. 150 184–150 202, 2019.

[2] N. Webb, "A fork in the blockchain: income tax and the bitcoin/bitcoin cash hard fork," *North Carolina Journal of Law & Technology*, vol. 19, no. 4, p. 283, 2018.

[3] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[4] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Generation Computer Systems*, vol. 107, pp. 841–853, 2020.

[5] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: a survey," *arXiv preprint arXiv:1908.08605*, 2019.

[6] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[7] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Gasol: Gas analysis and optimization for ethereum smart contracts," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 118–125.

[8] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against underpriced dos attacks," in *International Conference on Information Security Practice and Experience*. Springer, 2017, pp. 3–24.

[9] F. Bodon and L. Rónyai, "Trie: an alternative data structure for data mining algorithms," *Mathematical and Computer Modelling*, vol. 38, no. 7-9, pp. 739–751, 2003.

[10] A. Alimoğlu and C. Özturan, "Design of a smart contract based autonomous organization for sustainable software," in *2017 IEEE 13th International Conference on e-Science (e-Science)*. IEEE, 2017, pp. 471–476.

[11] C. Dannen, *Introducing Ethereum and solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Springer, 2017, vol. 1.

[12] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[13] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

[14] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "Smartinspect: solidity smart contract inspector," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 9–18.

[15] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[16] P. Hegedűs, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," *Technologies*, vol. 7, no. 1, p. 6, 2019.

[17] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[18] V. Valaštín, K. Košťál, R. Bencel, and I. Kotuliak, "Blockchain based car-sharing platform," in *2019 International Symposium ELMAR*. IEEE, 2019, pp. 5–8.

[19] G. Fenu, L. Marchesi, M. Marchesi, and R. Tonelli, "The ico phenomenon and its relationships with ethereum smart contract environment," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 26–32.

[20] F. Hartmann, X. Wang, and M. I. Lunesu, "Evaluation of initial cryptoasset offerings: the state of the practice," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 33–39.

[21] U. W. Chohan, "Initial coin offerings (icos): Risks, regulation, and accountability," in *Cryptofinance and Mechanisms of Exchange*. Springer, 2019, pp. 165–177.

[22] D. S. Demidenko, E. D. Malevskaia-Malevich, and Y. A. Dubolazova, "Iso as a real source of funding. pricing issues," in *2018 International Conference on Information Networking (ICOIN)*. IEEE, 2018, pp. 622–625.

[23] V. Buterin, "Bitcoin multisig wallet: the future of bitcoin," *Bitcoin Magazine, March*, vol. 13, p. 2014, 2014.

[24] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.

[25] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: Past, present, and future trends," *arXiv preprint arXiv:2005.14282*, 2020.

[26] K. Košťál, T. Krupa, M. Gembec, I. Vereš, M. Ries, and I. Kotuliak, "On transition between pow and pos," in *2018 International Symposium ELMAR*. IEEE, 2018, pp. 207–210.