A Survey on Smart Contract Automated Repair Methods

Ru Ji
r2ji@uwaterloo.ca
20987675

Abstract

Ethereum is one of the most popular topics in the blockchain community these days. The main reason is that it provides users with the availability to develop their own smart contracts. However, vulnerability comes along with the wild use and convenience provided by smart contracts. Vulnerabilities in smart contracts can lead to huge financial damages. At the same time, once the smart contracts are deployed on the Ethereum blockchain, they will be immutable, which means if the contract is flawed, there is hardly any method to compensate for that. Therefore, it is important to make sure any potential vulnerabilities are properly discovered and repaired before deployment. This paper firstly shows current research focusing on automated smart contract repair, then I will discuss the challenges and current problems, along with potential future on this topic.

1 Introduction

With the development of blockchain technology, cryptocurrency has seen explosive growth in recent years. By 12th Feb, 2022, there are 17,514 cryptocurrencies for trade from coinmarketcap [1].

As the first generation cryptocurrency, bitcoin [22] demonstrated the possibility to create a value-transfer system that is available across the world and free to use. However, bitcoin has its limitations. Due to the performance and scalability issues, bitcoin can not support complex applications. Bitcoin is mostly used as a cryptocurrency instead of a blockchain platform. Ethereum [6] was therefore proposed to provide the user with the possibility to create DApps (Decentralized Applications) by developing smart contracts. Ethereum has been regarded as the most creative blockchain technique after Bitcoin.

However, the availability and the financial behavior in Ethereum also attract the attention of attackers. There have been continuous attacks on smart contracts ever since Ethereum was launched, According to the study performed by Perez and Livshits, they surveyed 21,270 vulnerable contracts reported and found out that 504 of them have been subjected to exploits [24]. For example, the DAO (decentralized autonomous organization) was stolen 60$ million of ether due to the reentrancy vulnerability [19] in the smart contracts in 2016, leading the Ethereum community with the only choice of a hard fork.

With that said, in order to avoid financial loss, there is a need to make sure that the smart contracts are free of vulnerabilities before deployment, and there is also a need to design a pattern which can upgrade a deployed vulnerable contract. The most common way to deal with this problem is vulnerability detection tool, with source code or bytecode of a smart contract as input. However, even though the location of vulnerability and the potential exploit path are clearly pointed out, human coders might still not be able to correctly rectify the flawed contracts, or they can not do this work as accurate and efficient as an automated tool, and this is reason the design of automated smart contract repair come to sight.

There are several automated smart contract repair tools proposed, SCRepair [30], SMARTSHIELD [31], EVMPatch [26]. These tools share a pipeline pattern of detecting vulnerabilities first, then they implement different automated patching techniques to patch the vulnerabilities found by the detection tool, then use a test suite to test the performance of the tool. However, in this pipeline, there are still some open problems, and several steps can still get some improvements in the future research to improve the overall accuracy and scalability.

2 Background

2.1 Smart Contract and Bytecode

A smart contract is a collection of code and data that reside in Contract Account. They are typically written in high level languages such as Solidity[2]. They are used to implement arbitrary rules as well as guarantee to produce the same result for decentralized parties. Anyone can execute smart contract code by making a transaction request. Bytecode is the format in which smart contracts exist and execute, it is compiled from source code. The bytecode of smart contract is composed of three parts: *creation code*, *runtime code* and *swarm code*.

2.2 Ethereum Account and Transaction

Account is the most basic unit to identify an entity in the network. There are two types of accounts: *External Owned Account* and *Contract Account*. External Owned Account(EOA) works as the user in the Ethereum network. EOA has several abilities: it can transfer tokens, invoke deployed smart

contracts and store received tokens. EOA can also link itself to Contract Account, EOA can deploy a smart contract into a Contract Account. However, the deployment is achieved through a transcation. An account deploys a smart contract by sending a transaction that contains the bytecode of smart contract to address 0x0. After the contract is deployed, all accounts (both contract account and EOA) are able to invoke the smart contract via the contract address.

2.3 Ethereum Virtual Machine

Ethereum Virtual Machine (EVM) is a simple stack-based architecture, the function of the EVM stack is to store the results of intermittent execution of bytecode instructions. When the source solidity code is compiled into bytecode, EVM is able to provide a runtime environment for bytecode, at the same time, EVM manages the execution of transactions and the transit of blockchain state.

EVM specifies how the system state is altered given a series of bytecode instructions and a small tuple of environmental data, the detailed information can be checked from the yellow paper[28].

2.4 Gas Usage

Gas is a mechanism used in Ethereum to constrain external calls to smart contracts. It is the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. From the bytecode level, each instruction equals to a certain amount of gas, all the instructions calculated together, multiplied with the gas price, this is the simplified way to understand how to get the gas for the execution of a smart contract.

2.5 Symbolic Execution in Ethereum

Symbolic execution is a technique that uses constraint solving to explore a program's state space. Symbolic execution on EVM have several differences than traditional execution [21] . For example, the previously mentioned gas usage is a new mechanism in Ethereum blockchain, Ethereum also implements different memory and persistent storage models. Besides, transactions in Ethereum consist of a value and a data buffer, the data buffer contains information about the behaviors of this particular contract in execution. Symbolic execution of smart contracts involves symbolic transactions. Symbolic transactions are applied to all Ready states.

There have been several open-sourced symbolic execution tools in Ethereum, Manticore [21],KEVM[14], VerX[25], Mythril[3], these tools have also been used in commercial vulnerability detections.

3 Problem Definition

The goal of automated smart contract repair is to generate correct, high-quality, gas-optimized fixes for the vulnerable smart contract. The whole repair process can be described in the following formula:

$$(C, U, T, L) => C'$$

Notation In this formula, C represents a vulnerable smart contract, U represents a set of detected vulnerabilities, T represents a test suite, L is the maximum gas usage. C' is the output of the fixed repair.

Definition The successful vulnerability repair is defined as:

C' comes from C, but with all the vulnerabilities in U fixed, Passing all tests in T, and the Maximum gas usage of feasible execution paths <= L.

The smart contract repair problem introduces extra computational complexity, which is mainly due to the fact that smart contracts are typically running on the top of the blockchain systems that impose certain constraints on the total computational resources used by the contract. The rectified contract should also make sure that no further vulnerabilities are brought by the rectification process.

4 Motivation

Rodler et al conducted a developer study [26]. In this study, they asked 6 professional developers who consider themselves familiar with blockchain technologies but are not familiar with Solidity to manually patch three vulnerable contracts and patch these contracts using the tool EVMPatch. The result shows that none of the six developers were able to produce a correct upgradable contract, which shows that EVMPatch offers more efficiency, usability, and automation. On repairing smart contracts, there is a need to implement an automated repair tool other than manually repairing.

Weiqin et al. [32] also conducted an empirical study to analyze the current problems with solidity developers, and suggested some directions for developers and researchers.

5 Automated Smart Contract Repair

There is a general pipeline for automated smart contract repair problems. At first, vulnerability detection tools should be implemented to properly locate the vulnerabilities in the smart contracts. After detection, automated repair tools are implemented on these vulnerabilities that have been detected. The repaired contracts will then be passed into the test suite to check the accuracy of the repair tool. In different steps of this pipeline, there are novel techniques proposed by researchers. This chapter will elaborate on these novel techniques.

5.1 Vulnerability Detection

Input Vulnerability detection tool analyzes either the contract source code or its compiled EVM bytecode. Most tools target bytecode. The reason is that source rewriting may risk corrupting storage-layout, and make it harder to test with transaction data. However, source code analysis still provides the feasibility of more intuitive human analysis.

ZEUS[17] takes source solidity code as input, and first generates the abstract syntax tree(AST), then it designs a XACML-styled policy. ZEUS also provides the Solidity to LLVM bitcode translator, which can translate the policy and Solidity code to LLVM bitcode, where it implemented Horn clauses to check that the policy is respected.

Output Most tools will output the vulnerability type and location of vulnerability, such as Mythril[3], MAIAN[23], Securify[27], Slither[9]. Teether[18] is slightly different as it will generate an exploit to the vulnerability.

5.1.1 Fuzzing. There are also some fuzzing-based vulnerability detection techniques. Choi et al. [8] proposed SMARTIAN to use both static and dynamic Data-flow analysis. Ting et al.[7] proposed TokenScope to detect inconsistent behavior of currency tokens with a tool called TokenFuzzer. Grieco et al.[11] proposed Echidna, which supports assertion checking, custom property-checking, and estimation of maximum gas usage. Youngseok et al. [29] proposed Fluffy, which is a multi-transaction differential fuzzer to find consensus bugs.

Fuzzing can also be conducted from other perspectives. Ashraf et al. [5] proposed GasFuzzer which explores the effects of gas allowance manipulation to expose gas-oriented exception security vulnerabilities.

5.1.2 Symbolic Execution. There have been several open source Ethereum symbolic execution tools. Most of the vulnerability detection tools implement symbolic execution based techniques.

TEETHER [18] can automatically create and verify exploits for smart contracts, after constructing CFG, TEETHER generates critical paths and uses the SMT-solver Z3 to prune the search space and to compute multi-transactional exploits. MAIAN[23] symbolic executes the contract with Z3-solver, then after the detection, MAIAN attack the contract with concrete transactions to validate the detection results.

Securify[27] relies on Datalog. It firstly decompiles the bytecode into a static-single assignment form, then it represents the code as DataLog facts. In the last step, it will interpret the patterns to check whether they are compliant with each other.

MadMax[10] works similarly to Securify, it also implemented Datalog technique, the difference is that it focuses on vulnerabilities related to gas. It is the first tool to detect "unbounded mass operations". Securify 2.0 supports an updated version of the smart contract language and more types of vulnerability detection.

Oyente [20] takes the bytecode of a smart contract and a state of the Ethereum blockchain, then it simulates the EVM and explores different paths of the contract.

There has also been some research on formally verifying Ethereum smart contracts. Hirai et al.[15] use LEM to generate definitions for theorem provers such as Coq. Grishchenko et al.[12] presents a complete small-step semantics of EVM bytecode and formalizes it using the F* proof assistant.

5.2 Automatic Repair

Current research on this topic implemented various different techniques. Xiao et al. [30] proposed SCRepair which searches among mutations of the buggy contract, and it also considers the gas usage of the patch to select the most gas-friendly patch available. Zhang et al. [31] proposed a template based repair method which aims at three typical bugs, and SmartShield is at the same time gas-friendly, it only introduced 0.2% gas increment for each repair. EVMPatch [26] took one step further than SmartShield as it is more flexible and it proposed a solution to upgrade the vulnerable contract on the blockchain.

5.2.1 Repaired contracts deployment. Although once the smart contracts are deployed on the blockchain, it will no longer be able to edit or remove the contracts. There have been several ways to achieve the goal of contract upgrade, each with their own benefits and disadvantages.

- The first approach is to deploy the patched contract at a new address and migrate the state of the original contract to it. However, state migration is specific to the contract and must be manually implemented by the developers to have access to all the internal states of the old contract, and a procedure in the new contract to accept state transfers.
- To avoid state migration, developers can use a separate contract as a data storage contract(eternal storage pattern). The disadvantage of this approach is that this adds additional gas overhead, which means that every time the logic contract needs to access data, it must perform a costly external call into the data storage contract.
- **Proxy contract** The most favorable version of proxy-pattern is delegatedcall-proxy pattern. Once smart contract is split into two different contracts, the first one is an immutable proxy contract, which holds all funds and all internal states, and the business logic is in another contract, this one is completely stateless and implements purely the business logic. User enters the system through this proxy contract. The proxy contract will forward all function calls to the logic contract using the `DELEGATECALL` instruction, which gives the logic contract access to all internal states and funds stored in the proxy contract. When contract need to be upgraded, the upgraded contract is deployed on the blockchain, then its address is updated in the proxy contract. Using this approach saves the need to migrate data, and at the same time the upgrading process is transparent to users.

5.2.2 Template-based Repair. SMARTSHIELD implemented a template-based repair method. It firstly scans real-world smart contracts with three existing detection tools: Securify, Osiris, Mythril. The repair process consists of two phases, semantic extraction and contract rectification. In the first phase, SMARTSHIELD extracts bytecode-level semantic information from the unrectified EVM bytecode, then given the extracted bytecode-level semantic information, SMARTSHIELD fixes insecure control flow and data operations. The novel idea in SMARTSHIELD is that it proposed control flow transformation and DataGuard insertion. The former can be used to solve the bytecode inconsistency problem after rectification, while the latter one is used to prevent further vulnerabilities.

5.2.3 Gas Usage. Several tools have took gas optimization into consideration when they are implemented[30][31]. Integrating gas-awareness into the smart contracts repair technique is crucial. Because if the gas consumption after repairment is not monitored, unnecessary gas consumption can be excessive and lead to either financial loss or out-of-gas exceptions. No one wants a super secure contract but the cost is extremely high that one transaction will cost thousands of ETHs, and most of the cost is also totally unnecessary.

SMARTSHIELD achieved gas optimization by appending the secure functions to the end of the generated EVM bytecode, then replacing the insecure operation with a secure function invocation .The result shows that for each rectified contract, SMARTSHIELD inserted 43.6 instructions on average, and the average size increment of each contract is around 1.0%.

SCRepair chooses the mutant with lower gas usage to achieve gas optimization.

6 Open Problems and Potential Future

6.1 Non-Template Based Repair

Most of the automated repair tools mentioned in this paper implemented a template-based method. Even those methods which do not directly implement a template based method, the implicit logic is also template based. For example, The detection rule behind TEETHER is finding out contracts that lead to a payment to an arbitrary address. In reality, new types of vulnerability and attacks continue to appear and it takes effort to create a template for every newly discovered vulnerabilities. Current research also mostly only focuses on several famous vulnerabilities.

Although there are few deficiencies with template based repair methods, it is not easy to jump out of template based methods to design an approach which can be very general towards vulnerabilities. However, there exists a balance between template based and non-template based repair methods. In the future tool design, several things should be noticed. Firstly, creating new templates should be made easier. When new type of attacks or vulnerabilities are discovered, implementing them into the repair tool should be made easy. Secondly, the vulnerability detector can be extended so that it can explore more vulnerabilities which have not yet been discovered publicly.

6.2 Dynamic Detector

Currently, the automated repair tools mostly use static vulnerability detection tools. Most of them implement the same method: using symbolic execution and adding constraints on the behaviors which come from the templates. While static analysis itself is not the perfect approach, for example, some static analysis tools can result in false positive reports. Therefore, it is of importance to have some other testing method on top of it to make the result more accurate, and dynamic detector is one of them. There are some open source dynamic testing frameworks in Ethereum, such as truffle[4]. Besides, the aforementioned fuzzing approaches can be implemented as dynamic validation.

6.3 Apply More Convincing Test Suite

There are some deficiencies in using history transactions as the test suite. First of all, collecting these transactions is not a easy task, throughout these years Ethereum blockchain has become huge that it will take excessive time and space to sync the data. Secondly, not all contracts have effective history transactions, some contracts have no history transactions, and there is a possibility that the history transactions are unable to exploit the vulnerability. An idea towards this problem is to use self-constructed test suites which target specifically exploit paths.

6.4 Bytecode Rewriting

The bytecode level contract rectification may change addresses of instructions, for example, unaligned target addresses of instructions `JUMP`, `JUMP1` are required to be updated. SMARTSHIELD requires a complete control-flow graph to update jump targets and data references. EVMPATCH instead implements a trampoline based rewriting strategy which does not need an accurate CFG thus making it easier to scale to larger and more complicated contracts. However, both methods have deficiencies. Trampoline-based approach avoids fixing up any references with the cost of additional jump instructions, therefore extra gas cost. The simple approach SMARTSHIELD presents is limited to the templates they described, even within this approach, when the moved instructions depend on the execution result of `CALL`, there will be dependency conflict which SMARTSHIELD is unable to solve.

6.5 Other Chains

This survey focuses on Ethereum, while other contract based chains also face similar problem as in Ethereum. EOSIO blockchain [16] is famous for its Delegated Proof-of-State. EOSafe [13] is a static analysis framework for vulnerability

detection in EOSIO WASM smart contracts. The framework of EOSafe is composed of a practical symbolic execution engine for Wasm, a customized library emulator for EOSIO smart contracts, and four heuristic-driven detectors to identify the four vulnerabilities. There is no research on automatically repairing EOSIO contracts now.

7 Conclusion

Current researches on automated smart contract repair share similar pattern. In the first step a detection tool is implemented to locate vulnerable point, then various different approaches are used to automatically repair the vulnerabilities. Besides correctly rectifying the vulnerable point, there are also several things to consider. Firstly, after the smart contract is modified, the tool should ensure that the rectification process will not cause other unexpected vulnerabilities. The gas usage should also be monitored so that the rectified code will not bring unaffordable gas increment.Though these tools discussed in this survey paper show excellent result, they still have their deficiencies and limitations. Some steps within the pipeline of automated repair tool can expect further research and improvements. For example, dynamic detector can be added after static analysis to deal withe false positive reports from static analysis. Regarding the evaluation of the repair tool, more convincing test suite can be introduced. Bytecode level repair approaches still have some deficiencies with bytecode rewriting, dependency conflict might occur. Further reasearch can focus more on how to properly solve this deficiency.

References

[1] 2022. Coinmarketcap. https://coinmarketcap.com/

[2] 2022. Documentation of Solidity. https://solidity.readthedocs.io/en/v0.8.12/

[3] 2022. Ofiicial site of mythril. https://github.com/ConsenSys/mythril

[4] 2022. Truffle offical page. https://github.com/trufflesuite/truffle

[5] Imran Ashraf, Xiaoxue Ma, Bo Jiang, and Wing Kwong Chan. 2020. GasFuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. *IEEE Access* 8 (2020), 99552–99564.

[6] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014).

[7] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1503–1520.

[8] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.

[9] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[10] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[11] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 557–560.

[12] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

[13] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. {EOSAFE}: Security Analysis of {EOSIO} Smart Contracts. In *30th USENIX Security Symposium (USENIX Security 21)*. 1271–1288.

[14] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Roşu. 2017. *Kevm: A complete semantics of the ethereum virtual machine*. Technical Report.

[15] Yoichi Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*. Springer, 520–535.

[16] Yuheng Huang, Haoyu Wang, Lei Wu, Gareth Tyson, Xiapu Luo, Run Zhang, Xuanzhe Liu, Gang Huang, and Xuxian Jiang. 2020. Characterizing eosio blockchain. *arXiv preprint arXiv:2002.05369* (2020).

[17] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[18] Johannes Krupp and Christian Rossow. 2018. {teEther}: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1317–1333.

[19] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[20] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[21] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[22] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[23] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*. 653–663.

[24] Daniel Perez and Benjamin Livshits. 2019. Smart contract vulnerabilities: Does anyone care? *arXiv preprint arXiv:1902.06710* (2019), 1–15.

[25] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677.

[26] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2021. {EVMPatch}: Timely and Automated Patching of Ethereum Smart Contracts. In *30th USENIX Security Symposium (USENIX Security 21)*. 1289–1306.

[27] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

[28] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014),

1–32.

[29] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 349–365.

[30] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. 2020. Smart contract repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–32.

[31] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 23–34.

[32] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2084–2106.