Software Engineering Practices in Smart Contract Development: A Systematic Mapping Study

Antonios Giatzis[1], Elvira-Maria Arvanitou[1], Danai Papadopoulou[1], Theodoros Maikantis[1], Nikolaos Nikolaidis[1], Daniel Feitosa[2], Christos Georgiadis[1], Apostolos Ampatzoglou[1], Alexander Chatzigeorgiou[1], Evdokimos Konstantinidis[3], Panagiotis Bamidis[3]

[1] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
[2] Institute for Mathematics, Computer Science and AI, University of Groningen, Netherlands
[3] School of Medicine, Aristotle University, Thessaloniki, Greece

Abstract. Smart Contracts are pieces of software that are deployed in Blockchain infrastructures to enable the interaction (and production of value) between unknown parties, without intermediaries, but in a trustworthy and transparent manner. A key to Smart Contracts' success is their delivery to excellent standards of quality (e.g., security, documentation, code understandability etc.). To achieve this goal, the development of Smart Contracts needs to be driven by proven software engineering practices. In this paper, we conducted a systematic mapping study to get a comprehensive overview on how "*good*" software engineering practices are applied to Smart Contract Development. To identify primary studies that lie on the intersection of software engineering and smart contract development, we have selected specific publication venues and queried the literature. After applying the selection criteria, 113 studies were identified, analyzed, and synthesized results have been reported. The results provided some actionable implications for researchers and practitioners.

Keywords: Smart Contract Development, Blockchain, Software Engineering Practices, Software Quality, Cost, Mapping Study.

1 Introduction

After the introduction of Bitcoin in 2008 [6], the technology of Blockchain (BC) has risen enormously, not only in the field of cryptocurrencies, but also for serving application domains that yield safety and validation. BC is a decentralized peer-to-peer network, whose users are called nodes and perform transactions using cryptographic algorithms (stored in a shared ledger for assuring transparency). Along with the rise of BC, Smart Contracts (SCs) [9] began to popularize. SCs are computer programs that exist inside a blockchain network, imitating physical contracts, triggered and executed when certain, predefined conditions are met.

From a technical perspective, the execution cost of a SC is related to the size and complexity of the source code [2], with the size, depending on the SC complexity, usually having hundreds of lines of code [11]. A key distinction of smart contracts, compared to other software, is that their code cannot be maintained after deployment

to preserve trustworthiness. Since it is difficult to fully test the source code before deployment [13], errors are often discovered after the contract is executed, which can undermine trust from the end-user's perspective. Consequently, there is a clear need to apply “good” software engineering (SE) practices to improve the structural quality of SCs. An additional key consideration in SC development is the effective management of associated costs. As a general principle, as the codebase increases in size and complexity, the cost of executing a transaction within a blockchain network (referred to as gas) also rises. Since gas consumption is a critical factor in the success of smart contracts, this growth in size and complexity can negatively impact a contract’s competitiveness. However, such costs must be collectively considered alongside the cost of developing less “gas hungry” SCs. Based on this, we can deduce that there is a need to introduce a collective cost model for the complete SC lifecycle (need-2). To address these needs, we conducted a Systematic Mapping Study (SMS) aiming at the following goals: (g1) investigate SE practices currently used in SC development (addressing ***need-1***) and (g2) provide a list of costs for the entire lifecycle of a smart contract, including both development and operation costs (addressing ***need-2***).

2 Related Work

Vacca et al. [12] conducted a systematic literature review (SLR) to identify current problems and solutions in BC and SC development. They presented challenges related to SC testing, code analysis, metrics, security, DApp performance and BC applications. After the selection process, the authors analyzed 96 articles written from 2016 to 2020. One of their most important outcomes was the importance of bug detection in SCs. Demi et al. [3] carried out a SMS to identify the uses of BC in SE and its contribution to the SE landscape and categorized their results into eight areas: SE process and management, software requirements, testing, quality, maintenance and configuration management, and professional practices. We note that their study excluded papers on SE issues of BC-based software and included only the vice versa.

Alharby et al. [1] performed a SMS on BC based on SCs. The aim of this work is to identify and classify all peer reviewed research that has been conducted on SC technology. The search process was conducted up to 2018 using Scopus, ACM, Wiley, IEEE, and Springer, identifying 188 primary studies. They classified the outcomes into six categories: security, privacy, software engineering, application, performance and scalability. They also derived that 21% of the papers’ topics were about SE, noting that the number has increased by approximately eight times, compared to 2017. Macrinici et al [5] conducted a SMS to determine the channels for publishing SC-related research. After applying the selection criteria, 64 studies were identified. Based on their results, they managed to divide the papers into the four categories: security, privacy, scalability of BC and programmability of SCs. Regarding programmability of SCs, they identified security and reliability as the major key challenges as well as the economical challenge in association with gas cost.

Sanchez-Gomez et al. [8] performed a systematic literature review on the Software Development Life Cycle of SCs. More specifically, the authors focused on reviewing model-based software design and testing in SC. The results suggest that there is no established methodology for software validation or development of SCs. Additionally, they emphasized the need for efficient SC testing as it is the only way to reduce

functionality, security, and performance risks. Moreover, they highlight the importance of model-based design and testing as these reduce errors in design and code.

Tariq and Colomo-Palacios [10] conducted a SMS related to the interaction between SC and SE. The aim of this study was to identify opportunities and potential problems that may arise in SE while using SCs. The search process was applied on five DLs (ACM, IEEE, ScienceDirect, Springer, and Wiley), and resulted in 8 primary studies. The results indicate a shortage of professionals capable of ensuring security during SC development. At the same time, smart contract mechanisms can facilitate trust between untrusted parties. Finally, Kannengiesser et al. [4] conducted a SLR to identify challenges and solutions in SC development. They identified 29 challenges and 60 solutions from the literature, leading to them proposing 20 software design patterns, specialized for SC development in collaboration with SC developers.

3 Study

In this section, we describe the protocol of our systematic mapping study, which has been according to the guidelines proposed by Petersen et al. [7].

3.1 Study Design

The primary goal of our study, is to ***analyze*** software engineering literature for the ***purpose*** of characterization ***with respect to*****:** (a) the use of SE methods and tools on SC development; (b) the costs that associated with SC. We defined the following research questions:

RQ_1: *What SE practices are used in SC development?*

RQ_1 refers to the research goals that are being set related to software engineering practices in SC development. The goals have been retrieved from the research questions of each primary study. For the primary studies that do not provide a specific RQ, the goal is retrieved from the abstract of the primary study. By answering this research question, we provide the most used SE practices that are used for the implementation of SC.

RQ_2: *What are the costs involved when developing SCs?*

RQ_2 is related to how researchers approach their studies when measuring gas consumption or determining execution costs in general. By answering this research question, we shed light if there are formulas of the different types of costs and the methods for the reduction of these costs.

In order to answer these questions, we used well known practices and steps [7], which can be found online. Upon the assessment of 394 papers, we have retained 113 primary studies, which can also be found online [1].

3.2 Results and Discussion

SE Practices in SC development (RQ_1)**:** In this section, we present the most common SE practices that are used for SC development—see Table 1. The consolidated goals

[1] https://users.uom.gr/~a.ampatzoglou/aux_material/profesSupplementaryMaterial.zip

present the SE term that has been retrieved from the open card sorting methodology, while the number represents the studies in which the term is reported (30 more terms with lower frequency are omitted due to space limitation).

Table 1. SE Practices and Terms

Consolidated Goals	Count	Consolidated Goals	Count
Vulnerabilities	40	Cost Analysis	13
Security	33	Compilers / Languages	12
Testing	22	Code / Test Generation	9
Bugs / Defects / Faults	20	Code Clones	7
Gas	18	Software Design	6
Static Analysis	16	Debugging / Bug Fixing	5
Software Patterns	14	Framework	5

Based on our findings, an interesting result is that the frequency of the top 10 goals from the primary studies are quite high in number (197 times) in contrast to the rest 34 goals (102 times). We observed by their goals frequency that most primary studies focus on vulnerabilities (35%), security (29%), testing (19%) and bugs / defects / faults (18%). This finding is intuitive in the sense that SCs need to be reliable and trustworthy, given the fact that they intend to manage monetary transactions and have legal entities.

The high number of primary studies (40) mentioning the need to understand and mitigate **Vulnerabilities** in smart contract codebases highlights the researchers' focus on addressing these issues through various methods. Risks associated with errors during the coding phase, logical loopholes that can transform into code loopholes and weaknesses resulting in potential exploits and financial losses, are some of the problems that researchers have pointed and addressed as important. Researchers have developed and recommended various tools and methodologies to mitigate vulnerabilities in smart contracts and ensure their reliability at the pre-deployment stage. These include fuzzing tools [PS67], [PS96], [PS37] and static analysis tools [PS101], [PS30], tools for fixing insecure code patterns and securing bytecode before deployment [PS110], analyzing correlations between certain types of smart contracts and specific vulnerabilities to help developers avoid risky codebases [PS36], and emphasizing the importance of using the latest compilers with patched vulnerabilities [PS35].

Moreover, the emphasis on **Security**, which is mentioned in 33 of the primary studies, confirms the existing attention shown by the scientific community regarding issues related to the importance of safeguarding the codebase of the SC against malicious attacks, while at the same time, protecting the assets or functions that are ruled by these contracts. To address these challenges, primary studies propose security methodologies that practitioners can follow throughout the smart contract development lifecycle, identifying common problems and offering potential solutions [PS87]. These include frameworks that improve security by recommending and validating patterns and secure programming standards, helping developers implement secure contracts more efficiently [PS76]. Additionally, semi-automated or automated tools are

introduced to assist in detecting security issues before deployment on a live blockchain network [PS55], [PS37], [PS101], [PS64].

Third in our findings, **Testing** is mentioned as a goal in 22 primary studies, and it shows the recognition of the vital importance that testing methods play during the development lifecycle of a smart contract. Mossberg et al. [PS65] developed Manticore, a dynamic symbolic execution framework that can analyze simultaneous multiple smart contracts for discovering code bugs and verify code invariants and works by not making any hypothetical assumptions regarding the underlying execution model of the contract. Nguyen et al. [PS67] developed sFuzz, an automated fuzzing testing tool in which a generated execution trace of a smart contract is tested, and, through high code coverage and pattern analysis, any potential vulnerabilities can be discovered. A similar fuzzing tool was proposed by [PS108] and by simulating blockchain behaviors, it generated exploits such as unchecked transfer value and vulnerable access control. Its evaluation through 45,308 smart contracts brought up 554 exploitable cases, of which 306 have not been generated by any exploit generation tool until then. Ma et al. [PS63] created the tool Pluto that can detect three types of vulnerabilities under inter-contract scenarios: integer overflow, timestamp dependency, and reentrancy. When applied their tool on 39,3443 smart contracts, it has confirmed 451 previously unknown vulnerabilities, including 36 inter-contract vulnerability scenarios.

Focusing on the goals of **Bugs / Defects / Faults** (20 studies) and **Gas** (18 studies), the researchers emphasize the challenges associated with coding errors, logical flaws and under-optimized smart contracts, resulting in unexpected code behaviors and higher transaction costs, thus impacting the usability and cost-effectiveness of the smart contracts. To prevent the identification of bugs / defects / faults, extensive testing is required. In that sense, this line of research is quite active. Regarding smart contract bugs, J.F. Ferreira et al. [PS26] introduced SmartBugs, a framework for analyzing Solidity smart contracts and detecting defects. Zhang et al. [PS107] developed a dataset of 49 smart contract bugs, outlining the criteria for their detection and including examples of contracts with integrated bugs to avoid. Gao et al. [PS28] created SmartEmbed, a tool that uses structural code embedding to detect bugs, while the ĐArcher tool [PS109] was designed to identify synchronization bugs between the on-chain and off-chain layers of DApps. Regarding the higher transaction costs that the developers and the users might encounter, from the 2017 [PS15] it was apparent that under-optimized smart contracts cost more gas and that could be avoided by following gas-efficient programming patterns. Fuzzing has been proposed as a method for estimating the gas required by a smart contract [PS81], alongside other techniques to predict gas costs and prevent out-of-gas exceptions [PS44]. By analyzing a sample of 68,000 smart contract interactions over three years, Severing et al. [PS78] identified three primary cost drivers: external code, storage, and transaction base fees. Additionally, deployment gas usage has become increasingly costly due to programmers' assumptions about how their code would operate, which often results in unexpected gas consumption.

***What are the Costs Involved when Developing SCs? (RQ2)*:** During the lifecycle of the development of a smart contract, from the first line of code till its deployment on blockchain, there are certain types of costs that are involved, each with a distinct purpose and contributing: (a) on the quality, functionality, and security of the contract;

and (b) to the overall cost of a project. The early management and consideration of these costs become even more important by considering the immutable nature of a Smart Contract, which prevents gas optimization after deployment [PS15], [PS81], [PS44], [PS3], [PS21]. Based on our findings, the SC lifecycle costs can be divided into three main categories:

- **Development cost**: involves the necessary expenses (mostly personal effort) required for the programming of a smart contract codebase, ensuring all good practices are present. Delivering the SC at the required levels of quality.
- **Deployment Cost** involves the necessary fees that must be paid when a Smart Contract (or a bunch of Smart Contracts) is deployed onto a blockchain network and thus making it available for use.
- **Transaction Cost** corresponds to the necessary fees that must be paid for interacting with the functionality of the Smart Contract for executing transactions.

Researchers highlighted various challenges in SC development in terms of gas optimization and suggested security analysis tools and code patterns, actions to be taken to mitigate these challenges, aiming on reducing these costs and enhancing the overall quality of smart contract code.

In terms of **Tools**, Chen et al. [PS15] introduced GASPER with the main purpose to locate 7 expensive gas patterns, such as dead code and expensive loops operations, by analyzing smart contract's bytecodes and replacing them. From a sample of 4,240 real smart contracts, 80% were suffering from 3 (out of the 7 studied) gas patterns. Yu et al. [PS104] championed that reducing the number of opcodes could lead to less bytecode and eventually lowering the deployment costs. SCRepair was designed to detect vulnerable contracts and automatically repair them by generating correct code patches. Other research efforts focused on developing tools, such as the GASTAP that uses a control flow graph to estimate the functions calls' gas bounds [PS3]. Additionally, Di Sorbo et al. [PS21] collected 19 code smells produced by low efficiency of data storage and function implementation and developed the GASMET tool that evaluates the code quality of a smart contract from a static point of view, focusing on the overall functionality wastage in term of gas. Moreover, Albert et al. [PS4] developed a framework that automatically identifies patterns that are expensive by focusing on the stack bytecode operations. Finally, Xi and Pattabiraman [PS98] proposed GoHigh, a tool that replaces low-level function-related vulnerabilities with secure alternatives, resulting in more than 5% lower gas cost from a dataset of over 2,100,000 real-world smart contracts.

In terms of **Code Patterns**, various studies pointed out the necessity of creating a gas-optimized code-base free of bugs, while at the same time, any anti-patterns should be avoided. Chen et al. [PS12] defined 20 contract defects, categorized them into five categories (i.e., security, availability, performance, maintainability, and reusability), provided example code with the defects (e.g., anti-pattern: "*calling external smart contracts often or nesting calls into a loop"*), accompanied by possible solutions. Li [PS44] proposed solutions for the off-chain storage. For example, a solution for reducing the amount of gas used would be to use local variables instead of global ones. This is expected to lower the amount of the fees that are paid across several transactions. Additionally, Wang et al. [PS90] proposed patterns that involve batching. Finally, Zarir et al. [PS105] suggested that loops with dynamic boundaries could potentially lead to

higher gas cost drivers, with the risk of creating a transaction from a user that uses all the available gas. Some of the solutions they suggested is that the developers should know that the transactions usually take form by prioritizing for completion the ones with high gas price.

4 Implications to Researchers and Practitioners

Based on the findings of our study, several implications to researchers and practitioners can be highlighted. Regarding researchers, we propose the need for enhancing smart contract security by developing comprehensive security frameworks and innovative approaches for automated vulnerabilities detection. Additionally, the emphasis on testing shows that a need for safeguarding the codebase of the smart contracts against malicious attacks is a crucial task and researchers should examine new testing paradigms and expand the existing ones, focusing on tackle any inefficiencies that the old methodologies may possess, including automated testing tools and environments that simulate real-world exploits (and possibly the creation of up-to-date database). Furthermore, the need for a holistic approach for detecting and mitigate bugs is also a critical area of study that can be achieved through newer formal verification techniques and the specifically tailored combination of elements from both static and dynamic analysis. Finally, researchers could concentrate on developing predictive models for gas usage and execution costs as well as gas anti-patterns detection and repairing tools.

Regarding practitioners, this study has pointed out that the creation of under-optimized smart contracts during the lifecycle of the development of a smart contract can lead to unexpected code behaviors and higher transaction costs. Thus, impacting the usability and cost-effectiveness of the smart contracts. Additionally, by considering that the field of smart contract development is rapidly evolving, and new tools and methods are being suggested, the need of the adoption of proven SE practices it is even more imperative and necessary. The implementation of robust risk and cost management strategies is mandatory, including: (a) comprehensive and rigorous testing protocols in simulated environments for addressing on the early stages of production any potential security issues, (b) reusing components that are field-tested and can reduce development and cost during the development lifecycle. Staying informed with the latest research finding and incorporating well-established SE practices should be a standard part towards a secure and cost-effective smart contract development workflow.

5 Conclusions

This paper aims to provide an overview of the main SE practices and concepts that have been used in the literature of SC development. Special emphasis has been placed on the costs that appear in the complete lifecycle of a SC, and the ways to mitigate them effectively. The paper attempts to summarize the main trends in research and focuses on discussing the most prominent SE-related terms, as well as the main strategies for cost reduction. Through our systematic mapping study, it emerged that a significant portion of research focuses on the areas of vulnerabilities, security, testing, bugs and

gas consumption, topics that cover vital issues that arise during the development lifecycle of a SC and emphasize the demand for the creation and usage of robust security and testing protocols. In this way it will ensure the reliability and trustworthiness of a SC's code and logic.

Acknowledgments. This study has been partially funded by the Horizon Europe Framework Programme of the European Union under Grant agreement no 101058479.

References

1. Alharby, M., Aldweesh, A., Moorsel, A. v: "Blockchain-based Smart Contracts: A Systematic Mapping Study of Academic Research (2018)", International Conference on Cloud Computing, Big Data and Blockchain (ICCBB), Fuzhou, China, pp. 1-6 (2018).
2. Ajienka, N., Vangorp, P., Capiluppi, A.: "An empirical analysis of source code metrics and smart contract resource consumption", Journal of Software: Evolution and Process, 32 (10) (2020).
3. Demi, S., Colomo-Palacios, R., Sánchez-Gordón, M.: "Software Engineering Applications Enabled by Blockchain Technology: A Systematic Mapping Study", Applied Sciences, 11(7) (2021).
4. Kannengiesser, N., Lins, S., Sander, C., Winter, K., Frey, H., Sunyaev, A.: "Challenges and Common Solutions in Smart Contract Development", Transactions on Software Engineering, 48 (11), pp. 4291-4318 (2022).
5. Macrinici, D., Cartofeanu, C., Gao, S.: "Smart contract applications within blockchain technology: A systematic mapping study", Telematics and Informatics, 35 (8) (2018).
6. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System, https://bitcoin.org/bitcoin.pdf, last accessed 2024/08/04.
7. Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M.: "Systematic mapping studies in software engineering", 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08), Bari, Italy, pp. 68-77, 26 - 27 (2008).
8. Sánchez-Gómez, N., Torres, J., Garcia-Garcia, J.A., Escalona, M.J, Gutierrez, Javier J., Escalona, M.J.: "Model-Based Software Design and Testing in Blockchain Smart Contracts: A Systematic Literature Review", IEEE Access, 8 (2020).
9. Szabo, N.: Formalizing and securing relationships on public networks. First Monday (1997).
10. Tariq, F., Colomo-Palacios, R.: "Use of Blockchain Smart Contracts in Software Engineering: A Systematic Mapping", Computational Science and Its Applications – ICCSA (2019).
11. Tonelli, R., Destefanis, G., Marchesi, M., Ortu, M.: "Smart Contracts Software Metrics: a First Study", PLoS ONE, 18 (4) (2023).
12. Vacca, A., Di Sorbo, C., Visaggio, A., Canfora, G.: "A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges, Journal of Systems and Software, 174 (2021).
13. Zou, W. et al.: "Smart Contract Development: Challenges and Opportunities", Transactions on Software Engineering, 47 (10), pp. 2084-2106 (2021).