Towards a Secure Blockchain Ecosystem: Current Vulnerabilities and Future Directions in Smart Contract Security

Boya Zhao*

School of Computer and Information, Hebei Finance University, 071000 Hebei, China

Abstract. As a distributed shared transaction ledger, blockchain technology has the characteristics of decentralization, immutable, irreversible and traceable, and is changing the inherent model of traditional industries. Smart contracts, as one of the core applications of blockchain technology, provide the basis for a variety of practical applications. However, the frequent security problems of smart contracts have not only caused huge economic losses, but also hindered the development of blockchain systems. According to relevant studies, the economic losses caused by smart contract security breaches have exceeded billions of dollars. Therefore, the security of smart contracts has become a hot topic at home and abroad. This study discusses the technical vulnerabilities and corresponding solutions of smart contracts from the code level. In-depth analysis of security vulnerabilities in smart contracts can help identify and repair potential security hazards, improve the overall security of contracts, and prevent the theft of funds or abnormal execution of contracts. By raising the security awareness of developers, enterprises and users, it is possible to promote the application of smart contracts in high-security fields such as finance and law and promote the healthy development of the blockchain ecosystem.

1 Introduction

Blockchain technology [1, 2] is essentially a distributed shared transaction ledger that is jointly maintained by all nodes in the blockchain network under the constraints of a consensus protocol [3]. Blockchain technology has the characteristics of decentralization, tamper-proof, irreversible and traceable, which has changed the inherent mode of traditional industries.

Smart contracts [4], as one of the core applications of blockchain technology, provide the basis for a variety of practical applications. The concept of smart contracts was proposed by Szabo in 1994 [5], but it was not until the emergence of blockchain technology that it was provided with a reliable execution environment and application basis. However, the frequent occurrence of smart contract security incidents has not only caused huge economic losses, but also hindered the development of blockchain systems. According to Bcsec and Slowmist [6, 7], the economic losses caused by smart contract security breaches have exceeded billions

* Corresponding author: BOY2151520@maricopa.edu

of dollars. Therefore, the security of smart contracts has become a hot topic at home and abroad. The security of smart contracts can be analyzed at three levels: the solid code layer, the Ethereum Virtual Machine (EVM) execution layer, and the blockchain system layer.

The current research on vulnerabilities in smart contracts is of great significance, mainly reflected in the following aspects: As one of the core applications of blockchain technology, smart contracts are often used in key fields such as financial transactions and asset management. Studying vulnerabilities can help identify and fix potential security vulnerabilities, thereby improving the overall security of the contract and preventing the theft of funds or abnormal contract execution.

At the same time, the operation of smart contracts often depends on the accuracy and security of the code. Through in-depth study of vulnerabilities, users can enhance their trust in smart contracts and promote their application in more scenarios, especially in finance, law and other fields with high security requirements. This contributes to the healthy development of the entire blockchain ecosystem, promoting the design and development of more secure smart contracts, while the relevant education and training can be better disseminated and increase the security awareness of developers, enterprises and users. This will push more people to pay attention to the security of smart contracts and take appropriate precautions during the development process.

On the basis of the existing research results, this paper discusses the technical vulnerabilities faced by smart contracts at the code level and the corresponding solutions. Combined with deep learning technology, it looks forward to the future research direction, including the development of automated detection tools and the design of new protection mechanisms. The solutions of these vulnerabilities have different advantages and disadvantages in the types of vulnerabilities that can be detected, accuracy, detection time, etc., and also have their own limitations and improvement directions.

2 Vulnerabilities in smart contracts

2.1 Reentrant attack

An attacker repeatedly enters a function of a smart contract by calling an external contract, resulting in an abnormal contract state. On blockchains like Ethereum, contracts can call functions of other contracts. When one contract (contract A) calls another contract (contract B), the functions of contract B can be executed in the context of contract A. Smart contracts, on the other hand, usually update their status before making an important operation, such as a withdrawal. If an external contract is allowed to be invoked before a status update, an attacker can exploit this. During the attack, the attacker deploys a malicious contract (contract C) and designs a callback function in the contract. The attacker sends A transaction to contract A, triggering a withdrawal operation. Contract A calls contract C's callback function before making a withdrawal. In the callback function of contract C, the attacker calls the withdrawal function of contract A again, thereby re-entering contract A. If contract A allows another withdrawal before the status update, an attacker can withdraw funds multiple times, causing losses. the Ethereum reentrant vulnerability was caused by this mechanism, resulting in the most famous smart contract security breach of all time (the DAO attack [8]), which not only lost nearly $60 million worth of ether, but also directly led to the hard fork of Ethereum [9].

In terms of how this vulnerability is addressed, major exchanges currently check conditions in contracts, then update status, and finally interact with external contracts to ensure that even if a reentrant attack does occur, it will not affect status. Or in Ethereum, for

example, you can use ReentrancyGuard [10] (such as the nonReentrant modifier in the OpenZeppelin library) to ensure that a function cannot be reentered during a call.

2.2 Integer overflow error

Smart contracts typically update their status before making an important operation, such as a withdrawal. If an external contract is allowed to be invoked before a status update, an attacker can exploit this.

In Ethereum smart contracts, integers are generally specified as fixed-size and unsigned integer types, that is, integer variables can only be values within a certain range, beyond which integer overflow errors will occur. Solidity's integer variable steps are generally increments of 8 and support ranges from uint8 to uint256. Take uint8 as an example, its variable length is 8 bits and the range of numbers supported by storage is [0,255]. If you try to store data larger than this range into a uint8 type variable, the Ethereum virtual machine (EVM) [11] will automatically truncate the high value, resulting in an abnormal result and an integer overflow error. Unlike other programs, the losses caused by the smart contract integer overflow vulnerability are huge and irreparable. In 2018, the batchTranfer function in ERC20 was used in Meisoft [12]. In the implementation of this function, there is a danger of overspill, and attackers took advantage of this vulnerability, resulting in huge economic losses.

For such overflows, existing precautions include the use of secure math libraries [13], specifying the type and scope, and code auditing. Using secure math libraries means using programming languages that provide secure math libraries, such as Solidity, which can automatically check for overflows and underflows and throw exceptions when they occur. When defining a variable, choose an appropriate type (such as a signed or unsigned integer) and set reasonable range limits. The contract code is also audited periodically to ensure that all arithmetic operations are checked to avoid potential overflows or underflows.

2.3 Time dependence

Some contract functions may rely on block timestamps, which an attacker could exploit if they were able to manipulate block generation times. It is well known that on blockchains like Ethereum, each block has a timestamp that represents the time that block was mined. The smart contract can access this timestamp through block.timestamp. Although the timestamp is set by the miner, the miner can manipulate the timestamp within a certain range. For example, a miner can choose the timestamp when mining a block in order to influence some contract logic that depends on the timestamp. If there is a time limit in the contract logic, such as when certain operations can only be performed within a certain time frame, an attacker can manipulate the timestamp to advance or delay the execution of the attack for his own purposes. Assuming that the contract allows for a withdrawal after a certain point in time, an attacker could manipulate the timestamp to make the withdrawal take place ahead of time. For example, a contract is designed to release funds after a certain time. If attackers can influence block timestamps, they could cause contracts to release funds early, resulting in the loss of funds.

There are two main preventive measures: 1) Avoid using timestamps: Try to avoid relying on block timestamps in key logic, and instead use more reliable mechanisms (such as block height). 2) Introduce buffer time: Introduce buffer time in the operation involving time to ensure that even if the timestamp is manipulated, it will not pose a threat to the security of the contract.

2.4 Access control problems

Some features in a contract should be restricted to specific users or contract administrators. Improper access control can result in unauthorized users performing critical operations.

Access control is a mechanism for managing who can perform a particular action. In smart contracts, users' access to contract functions is often restricted through a system of roles or permissions, such as only the owner or administrator of the contract can perform certain key actions (such as withdrawal, modification of status, etc.). Two of the most common problems encountered by Ethereum virtual machines today include unrestricted access, where certain functions of a contract are not properly checked for permissions and can be called by anyone, potentially causing funds to be mistakenly transferred or the state of the contract to be maliciously changed. And permission inheritance, where multiple contracts may call each other in a complex contract. If permissions are not managed properly, attackers can use this chain to gain access to functions that are supposed to be restricted.

Such a vulnerability could lead to vicious attacks, such as malicious withdrawals, assuming that the contract allows anyone to call a withdrawal function that should in fact be limited to the contract owner. Attackers can withdraw money at will, resulting in the loss of funds. Or let's assume that some contracts allow the owner of the contract to upgrade the contract, and if access control is not implemented, an attacker could exploit the vulnerability to upgrade the contract and control the contract behavior.

The current solution mainly ensures that key functions restrict access with the modifiers onlyOwner or onlyAdmin [14], preventing unauthorized calls. Or with OpenZeppelin's AccessControl module [15], you can flexibly assign roles and permissions to ensure that different functions can only be performed by specific roles.

2.5 Transaction order dependence (front-running)

An attacker can manipulate the behavior of a contract by observing transactions in a trading pool and performing certain actions with their own trading priorities.

It is well known that in a blockchain network, all transactions to be processed are first put into a place called a "mempool" [16], from which miners choose to package transactions into a new block. Each transaction has its own gas fee [17], and usually the transaction that pays the higher fee is processed first. Users and attackers can view unconfirmed transactions in the transaction pool. This allows them to observe the trading intentions of other users, such as large transfers, purchases, bids, etc. When an attacker sees a profitable transaction (for example, a large purchase of a certain token), they can quickly submit their own transaction, paying a higher fee to ensure that their transaction is packaged before the target transaction. This early execution of trades is called "front-running".

Attackers exploit information asymmetries, where other users are not necessarily aware that their transactions affect market prices, or fail to respond quickly to their trading intentions. Through this, attackers can make improper profits. To take an obvious example: on A decentralized exchange (DEX), suppose user A submits a large purchase of a certain token, the price may rise as a result of the purchase. After observing this transaction, an attacker can first submit his own transaction, buy tokens at A lower price, and then wait for user A's transaction to execute, thus selling them at a higher price for a profit.

At present, the existing preventive measures include: the randomization mechanism is used to determine the execution order of transactions by introducing random elements, making it difficult for attackers to predict the execution time of transactions [18]. Before the transaction is confirmed, the transaction is randomly sorted, or a random number is generated using cryptography to determine the priority of the transaction. This can be done through algorithms in smart contracts. Of course, the use of privacy protection technology is also a

common means, such as zero-knowledge proof, hiding transaction information, so that attackers cannot observe the transaction content in advance. Or set a cooling time, such as after an important transaction is initiated, you need to wait for a period of time before you can perform the next action. This makes it impossible for attackers to respond quickly and submit transactions first.

3 Defense against Reentrant attack

This research focuses on the security of smart contracts and explores in depth the different types of vulnerabilities at the code level and their prevention measures. Through a detailed analysis of current smart contract vulnerabilities, it can be discovered that the performance of these major vulnerability detection means in practical applications, as well as the current technical limitations and challenges. These shortcomings and limitations can not only lead to serious economic losses, but also constitute an obstacle to the further development of blockchain technology. In particular, the DAO incident in the case of re-entry attacks has revealed the vulnerability of smart contracts in practice, and in this section, this paper will discuss the advantages and disadvantages of each defence in detail.

3.1 Defense against Reentrant attack

3.1.1 Check conditions and then update the status

Advantages: This approach ensures that necessary conditions are validated before state updates, reducing the risk of state inconsistencies. This can effectively prevent the attacker from reentrant if the conditions are not met. And the logic is clear, easy for developers to understand, the implementation is relatively simple, can quickly integrate into the contract development process.

Cons: If developers leave out necessary condition checks in complex logic, they may still leave vulnerabilities. An attacker can exploit these omissions for a reentrant attack. Adding condition checks at the same time makes the contract code verbose, which affects readability and maintainability, especially in multi-layer nested logic.

3.1.2 Using ReentrancyGuard

Advantages: Using the ReentrancyGuard library [19] provides a simple and effective way to prevent reentrant attacks and using the nonReentrant modifier can be easily applied to contract functions, reducing the security burden on developers. By encapsulating the reentrant protection mechanism, developers can focus more on the core business logic of the contract and less on the security details.

Cons: In some legitimate cases, ReentrancyGuard may restrict reentrant calls, affect some legitimate business operations of the contract, may result in an actual user experience, and if other parts of the contract do not adopt the same protection measures, an attacker can still launch reentrant attacks through other paths, reducing overall security.

3.2 The status of defense means is not updated

3.2.1 Operating safety

Advantages: After first using the secure math library, the math library can automatically detect integer overflows and underflows, ensuring that an exception is thrown in the event of an overflow, thereby protecting the state of the contract. The use of these libraries can significantly improve the security of contracts and reduce the risk of economic losses caused by calculation errors.

Cons: Security checks introduce additional computational overhead, which can lead to reduced contract execution efficiency, especially in scenarios with high frequency calls. Such a call may lead to a decrease in the utilization of the system, and if the security inventory used by the user is in breach, it may lead to new security risks, and the developer needs to periodically review the security of the library.

3.2.2 Specify the type and scope

Advantages: This method can reduce unexpected errors, by selecting the appropriate data type and range when the variable is defined, it can effectively reduce the errors caused by type mismatch and ensure the correctness of the contract logic. Specifying the type and scope makes the code more readable and easier for other developers to understand and maintain the contract.

Cons: Strict data types and ranges can limit some of the capabilities of the contract, leading to difficulties in extending the contract. Developers can misjudge the type and scope required, resulting in logical errors that affect the security of the contract.

3.3 Time-dependent defense measures

3.3.1 Avoid using timestamp

Advantages: By not relying on block timestamps, it significantly reduces the chance that an attacker can manipulate the contract using time and improves the security of the contract. Not relying on timestamps set by miners can improve the stability of the contract logic and reduce the contingencies caused by time manipulation.

Cons: Avoiding timestamps altogether may affect certain features of the contract, such as timed triggered operations, which may limit the application scenarios of the contract. Other mechanisms need to be designed to replace timestamps, which can increase the complexity and development difficulty of the contract.

3.3.2 Importing buffer time

Advantages: The introduction of buffer time mitigates the impact of time manipulation to some extent, ensuring that even if the timestamp is manipulated, the security of the contract will not be threatened. Another benefit of introducing a buffer time is that it allows the contract to provide additional security during time-sensitive operations. By setting a buffer time, the contract can provide additional protection when performing time-sensitive operations such as the release of funds, transaction confirmation, etc. Even if an attacker is able to manipulate the timestamp, their operation is subject to delays, reducing the probability of a successful attack.

Cons: Similar to the problems encountered in the previous code audit, setting the buffer time can cause users to experience delays during operations, which can affect the user experience, especially in scenarios that require fast execution. The introduction of buffer time increases the complexity of the contract logic, causing the developers of programs and systems to design carefully to avoid introducing new problems.

3.4 Defense measures against access control problems

3.4.1 Ensure that key functions use modifiers

Advantages: By using modifiers such as onlyOwner [20], developers can easily control the access to key functions of the contract and improve the security of the contract. Use can make the code structure clearer, and modifiers can restrict who has the right to perform key operations in the contract, such as withdrawals, status updates, or contract upgrades. This reduces the attack surface for malicious users, thereby protecting the assets and state of the contract and making it easy for other developers to quickly understand the rights management logic.

Cons: In complex contracts, the use of modifiers can cause logical confusion and make it difficult to trace the origin of permissions, especially when multiple inheritance occurs. Once permissions are set in place, subsequent changes can be very difficult, resulting in contracts that are not flexible enough to adapt to changing needs.

3.4.2 Using the AccessControl module

Advantages: OpenZeppelin's AccessControl module [21] provides a flexible role and permission management mechanism to assign specific permissions to different roles, improving contract security. At the same time, the module can reduce the complexity of rights management, make the contract more convenient to update and maintain, and facilitate the future expansion.

Cons: Learning curve: For beginners, understanding and correctly implementing AccessControl can take some learning time, increasing the difficulty of development. Potential security risks: If roles and rights are improperly assigned, unauthorized users may obtain permissions for critical operations, causing security problems.

3.5 Transaction order depends on defense

3.5.1 Using the randomization mechanism

Advantages: The introduction of random elements can effectively reduce the attacker's predictability of the order of transaction execution, thus reducing the risk of preemptive trading. Random ordering can improve the fairness of trade execution and reduce the misconduct caused by the order of transactions.

Cons: Implementing randomization mechanisms can add complexity to the contract, and developers need to ensure that randomness does not affect the core logic of the contract. The randomization mechanism may bring additional computation and storage overhead, affecting the efficiency of contract execution.

3.5.2 Privacy protection technology

Advantages: By using privacy protection technologies such as zero-knowledge proof, transaction information can be hidden, and attackers cannot observe the transaction content in advance, reducing the risk of being attacked. Privacy protection technology can enhance the security of contracts and prevent manipulations caused by information asymmetry.

Cons: The implementation of privacy protection mechanism has high technical requirements, and developers need to have relevant professional knowledge, which increases the difficulty of development. Privacy protection technology may cause the efficiency of contract execution to decrease and affect the user experience.

3.6 Possible solutions to the defects in the above defense measures in the future

Introduce smart contract security audit tools [22], develop and popularize more smart contract security audit tools, and use automated tools to detect potential vulnerabilities and security problems in contracts. Use tools to perform static analysis of contract code to detect unused modifiers, permission control vulnerabilities, etc. Or conduct simulated attacks to test the performance of contracts under different attack scenarios and identify potential risks such as time-dependent and reentrant attacks.

Introduce standardization and best practices, develop industry standards and best practices for smart contract development, and facilitate the design and implementation of secure contracts. Develop security standards through industry organizations or open source communities to regulate the smart contract development process. Write and promote security development guidelines for smart contracts, covering common defenses and implementation methods.

By using off-chain computing and collaboration [23], part of the complex calculation and verification process is moved off-chain, and external services are used for security verification to reduce the complexity of smart contracts. Use off-chain computing to process complex logic and submit results on-chain for verification, reducing the burden of smart contracts. Improve security and reduce the risk of a single contract being attacked by multiple participants verifying operations together.

Develop a blockchain-based random number generation mechanism [24] and use blockchain characteristics to develop a secure random number generation mechanism to avoid attackers' prediction and manipulation of transaction order. Ensure fairness and unpredictability by generating random numbers through on-chain events or user participation. Cross-chain technology is used to obtain randomness from multiple chains to improve the security and stability of random numbers.

4 Limitations and challenges

4.1 Reentrant attack

Implementing these defenses can increase maintenance costs for contracts, especially in large projects where complex security logic needs to be continuously monitored and updated. When designing a contract, developers must consider multiple attack scenarios to ensure that defenses do not affect the normal function of the contract, which can lead to a more cumbersome design process. Such a setup can be challenging for beginners to understand and properly implement these security precautions, adding to the difficulty of learning and development [25].

4.2 Integer overflow error

High-quality audits require experienced developers, yet the relative lack of qualified security auditors in the market makes resource acquisition difficult. And even after an audit, contracts still need to be regularly maintained and updated, and continuous security is a long-term challenge. With the rapid evolution of blockchain technology, new attack methods continue to emerge, and defense measures need to be continuously updated to cope with emerging threats.

4.3 Time-dependent defense measures

Solutions to time-dependent problems can be confusing to users, especially when it comes to time-sensitive operations, who may not understand the need for buffer time. As technology changes, time-dependent risks may evolve, and precautions need to be constantly updated and adapted to meet new challenges. Secondly, as mentioned in the introduction of buffer time, the introduction of buffer time increases the complexity of contract logic, so it is necessary to consider a variety of factors comprehensively when designing contracts to ensure that the time-related logic is safe and does not affect the function of contracts.

4.4 Prevention measures for access control problems

In complex contracts, permission management can become extremely complex, making it difficult for developers to maintain and manage, increasing the likelihood of errors. Even with access controls in place, contracts still need to be audited periodically to ensure that the rights management logic is correct, and the audit process can be time consuming. With the development of the project, the demand for access control may change, and developers need to adjust the permission management strategy in time to avoid security risks.

4.5 Transaction order dependent defense means

Precautions can have an impact on user behavior, resulting in changes in the way users understand and operate transactions. When designing a contract, it is necessary to consider the synergies between the various precautions to ensure that the overall security is not compromised.

5 Conclusion

The security of smart contracts has become a hot topic in the research of blockchain technology. With the popularity of blockchain technology, smart contracts are increasingly widely used in many fields such as finance and law, and their security directly affects the foundation of trust in these fields. Research on smart contract vulnerabilities not only helps to identify and repair potential risks, but also provides important support for enhancing user trust and promoting application popularization.

By analyzing the existing vulnerabilities and their solutions, it can be recognized that while some precautions have been proposed and applied, more effective solutions to the evolving means of attack need to continue to be explored. For example, the application of deep learning technology in vulnerability detection has broad prospects, and automated detection tools can be developed in the future to achieve more efficient vulnerability identification and response mechanisms. In addition, strengthening relevant education and training to improve the security awareness of developers and users is also an important

measure to ensure the security of smart contracts. Future research should focus on building a more comprehensive security system and security mechanism to cope with rapidly changing technological means.

References

1. M. Swan. Blockchain: Blueprint for a New Economy. O'Reilly Media, Inc. (2015)
2. Z. Zheng, S. Xie, H. N. Dai, et al. Blockchain challenges and opportunities: A survey. Int'l Journal of Web and Grid Services (2018)
3. L. S. Sankar, M. Sindhu & M. Sethumadhavan. Survey of consensus protocols on blockchain applications. In: Proc. of the 4th Int'l Conf. on Advanced Computing and Communication Systems (ICACCS), IEEE (2017)
4. V. Buterin. A next-generation smart contract and decentralized application platform. Ethereum White Paper (2014)
5. L. W. Ouyang, S. Wang, Y. Yuan, et al. Smart contracts: Architecture and research progresses. Acta Automatica Sinica (2019)
6. N. Szabo. Smart contracts: Building blocks for digital markets. Journal of Transhumanist Thought (1996)
7. Bcsec. https://bcsec.org; Slowmist. (2018)
8. The DAO. https://www.investopedia.com/tech/what-dao/ (2024)
9. Hard-Fork. https://www.investopedia.com/terms/h/hard-fork.asp (2020)
10. Z. Zheng, N. Zhang, J. Su, et al. Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, pp. 295-306 (2023)
11. E. Hildenbrandt, M. Saxena, N. Rodrigues, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), IEEE, pp. 204-217 (2018)
12. BeautyChain integer overflow. https://etherscan.io/token/0xc5d105e63711398af9bbff092d4b6769c82f793d (2018)
13. SafeMath. https://docs.statechannels.org/contract-api/natspec/SafeMath (2020)
14. H. B. N. A. Leandro. A Survey of Security Issues in Smart Contracts. http://arXiv:2001.03452 (2020)
15. N. Ivanov, C. Li, Q. Yan, et al. Security threat mitigation for smart contracts: A comprehensive survey. ACM Computing Surveys, 55(14s), 1-37 (2023)
16. A. Kosba, D. Papadopoulos, C. Papamanthou, et al. MIRAGE: Succinct arguments for randomized algorithms with applications to universal zk-SNARKs. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 2129-2146 (2020)
17. A. Laurent, L. Brotcorne & B. Fortz. Transaction fees optimization in the Ethereum blockchain. Blockchain: Research and Applications, 3(3), 100074 (2022)
18. P. Praitheeshan, L. Pan, J. Yu, et al. Security analysis methods on ethereum smart contract vulnerabilities: a survey. arXiv preprint arXiv:1908.08605 (2019)
19. N. Deshpande & R. Vaghela. Demystifying Reentrancy Attacks on Smart Contracts Understanding Types and Mitigations. Towards Excellence, (1) (2024)
20. Y. Fang, D. Wu, X. Yi, et al. Beyond "Protected" and "Private": An Empirical Security Analysis of Custom Function Modifiers in Smart Contracts. In: Proceedings of the

32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1157-1168 (2023)

21. H. Liu & Y. Sun. Using My Functions Should Follow My Checks: Understanding and Detecting Insecure OpenZeppelin Code in Smart Contracts. In: 33rd USENIX Security Symposium (USENIX Security 24), USENIX Association, pp. 3585-3601 (2024)
22. A. I. Beksultanova & A. L. Tkachenko. Analysis tools for smart contract security. In: 2nd International Conference on Computer Applications for Management and Sustainable Development of Production and Industry (CMSD-II-2022), SPIE, 12564, 221-228 (2023)
23. W. Chen, Z. Yang, J. Zhang, et al. Enhancing Blockchain Performance via On-chain and Off-chain Collaboration. In: International Conference on Service-Oriented Computing, Cham: Springer Nature Switzerland, pp. 393-408 (2023)
24. G. C. Sirakoulis, K. Christodoulou, S. A. Chatzichristofis, et al. RandomBlocks: a transparent, verifiable blockchain-based system for random numbers (2019)
25. E. Hildenbrandt, M. Saxena, N. Rodrigues, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), IEEE, pp. 204-217 (2018)