Short Paper: DeFi Attack Detection via Outlier Alerts on State Variables

*Zeyu Liang[1], Antonio Ken Iannillo[1], Christof Ferreira Torres[2], Bahareh Parhizkari[1], Sebastian Banescu[3], and Joseph Xu[3]

[1] SnT, University of Luxembourg
[2] ETH Zurich
[3] Quantstamp, Inc

Abstract. A growing number of attacks against Decentralized Finance (DeFi) systems urge practical and effective countermeasures acting on the attacks. However, current solutions to smart contract attack detection rely heavily on the contract source code and have limitations, including slow analysis and a restricted variety of attacks. To address these limitations, we propose a novel approach to detecting attacks against DeFi by tracing transaction execution and modeling the inner value changes of state variables. Our approach is real-time, lightweight, independent of contract source code availability, and not restricted to specific types of attacks. Experiments showed that our approach is effective in detecting various attacks but also raises false negatives.

1 Introduction

Smart contracts are the backbone of decentralized applications (DApps), including Decentralized Finance (DeFi) systems enabling secure and transparent operation for financial activities without the need for intermediaries. However, the increasing popularity of smart contracts has also led to a growing number of attacks targeting them. Design flaws in smart contract platforms, programming languages, and smart contract code cause risks on DeFi systems to be attacked. As a result, the DeFi ecosystem requires practical approaches to detecting and analyzing incidents to improve its security and reliability.

The existing solutions and tools for attack detection in smart contracts [2–6, 8–12, 15, 16, 19–21] rely heavily on the contract source code or EVM bytecode and ignore the execution traces of transactions. These solutions often integrate bulky external modules, which can be slow and impractical for real-time analysis. Moreover, these solutions are usually restricted to specific attacks and depend on the contract source code, limiting their ability to detect attacks and incidents.

In light of these limitations, we propose an approach to detect attacks against DeFi in a real-time scene by tracing transaction execution and modeling the inner value changes of state variables of smart contracts during each transaction. The approach analyzes the traces of transactions, extracts the value changes of state

* Corresponding authors: `rsscno1@mailfence.com`, `antonioken.iannillo@uni.lu`

variables during these executions, models the trends of these value changes, measures the likelihood of each transaction being attack-involved, and alerts the strongly suspected ones. This approach is lightweight, independent of contract source code availability, and not restricted to specific types of attacks, as it focuses on capturing general effects caused by attacks and incidents.

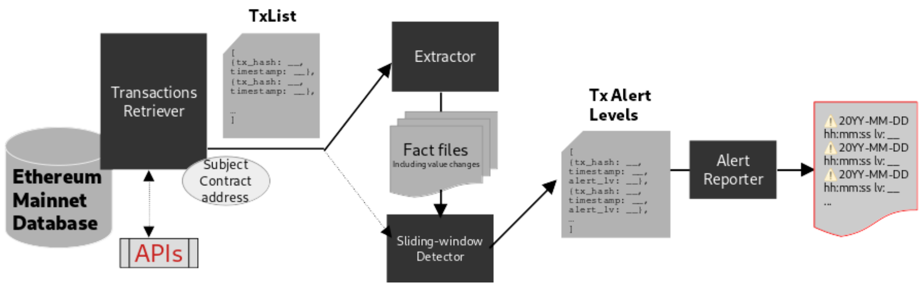
The proposed approach offers a practical and effective solution to detect and prevent attacks on smart contracts and improve the security and reliability of DeFi systems. We selected a dataset containing several typical DeFi systems with known attacks and performed experiments to test our approach, and it showed the power to detect a variety of attacks. However, in some cases, it also raises false negatives that indicate anomalies that are not directly associated with successful attacks (e.g., protocol changes, upgrades, or attempted attacks).

2 Background

Ethereum is a blockchain platform that allows developers to create and deploy smart contracts. *Smart contracts* are self-executing programs written in a programming language such as Solidity and stored on the Ethereum blockchain. They can be used to automate different kinds of transactions and have many advantages, such as eliminating the need for an intermediary and providing a secure and transparent way to conduct transactions. *State variables* are global variables where their state is persisted across transactions. They are declared outside of functions, and their values can be accessed and modified from any part of the smart contract. *Storage layout* defines the order in which state variables are persisted in Ethereum smart contract. Solidity organizes state variables according to the order they are defined in the smart contract. Each state variable is assigned a unique storage slot. Storage is organized as a key-value store where keys and values are 256-bit long. Solidity packs multiple state variables into the same slot if their type requires less than 256-bit. `SLOAD` and `SSTORE` are the opcodes in the Ethereum Virtual Machine (EVM) that handle the storage of values in a contract's state. SLOAD retrieves a value from storage and loads it onto the stack for processing, while SSTORE writes a value from the stack to the contract's storage. These two opcodes are crucial for the execution of smart contracts on the Ethereum network as they allow contracts to persist state between transactions.

3 Methodology

Our novel approach consists of four components: a transactions retriever, an extractor, a sliding window detector, and an alert reporter. The four components run in a pipeline while the Ethereum Mainnet keeps expanding, as shown in Figure 1. Given a target contract, the *transaction retriever* gets every transaction that interacts with the contract. The transactions drive the stream that is analyzed sequentially by the other components. For each transaction, the *extractor* generates a file (namely, facts file) with all the value changes of the contract's



```
[
{tx_hash: __,
timestamp: __},
{tx_hash: __,
timestamp: __},
...
]
```

```
[
{tx_hash: __,
timestamp: __,
alert_lv: __},
{tx_hash: __,
timestamp: __,
alert_lv: __},
...
]
```

Fig. 1. Pipeline of the proposed approach

state variables that happened in that transaction. The *sliding window detector* receives as input this file and produces an alert level for the transaction. The higher the alert level, the more likely it represents an attack. Finally, the detector feeds the *alert reporter* with the alert levels data to present them to the practitioner. The following subsections explain in detail each of these components.

3.1 Transactions Retriever

The first step is retrieving the transactions involving the smart contract under observation, which means that we do not consider only the transactions that targeted the contract (specified in the *to* field of a transaction) but also the transactions that triggered a message (internal transaction) to the contract.

Some online blockchain explorers (e.g., Blockchair [1]) provide API to fetch transactions (external and internal) on Ethereum. However, a modified Ethereum client can feed the extractor with a transaction hash identifier whenever the contract under observation is involved.

3.2 Extractor

The extractor gets the value changes of state variables in the transaction traces and to output these values at the beginning and end of each transaction.

The storage layout of the contract can show how the global variables are stored. One can easily derive the layout with the solidity compiler if the source code is available or via EVM decompilers if unavailable. The only difference would be that there may be no semantics associated with the global variables in the latter case. According to the Solidity documentation [13], state variables can also be stored as elements of a dynamic array or a mapping. Their slot index is hashed from the slot index of that array or mapping and other information, such as the variable's offset or key. Thus, our extractor saves the hashing key-value dictionary to decode the slot index of these complex cases.

For each transaction fed by the retriever, the extractor pulls its traces via an external Ethereum API (i.e., `geth.debug.traceTransaction`). It saves the contract address, slot index, and values of all the `SLOAD` and `SSTORE` opcodes operating on state variables. In the case of the slot index representing a dynamic array or mapping, the extractor decodes it according to the Ethereum storage layout. Finally, the extractor generates a fact file consisting of tuples $(C\_addr, slot, v_i, v_f)$, where C_addr is the contract address, $slot$ refers to the index of the storage slot or the dynamic array or mapping containing it, v_i is the value read at the first `SLOAD` for that slot, as its value at the beginning of the transaction, and v_f is the value written at the last `SSTORE` for that slot, as its value at the end of the transaction. There are some exceptional cases. If the first `SLOAD` occurred after the first `SSTORE` or there was no `SLOAD` on that slot index at all, the extractor should call another Ethereum debugging API (i.e., `geth.debug.storageRangeAt`) to get the correct initial value v_i for that transaction. If it still fails, the extractor sets v_i to $\perp$.

3.3 Sliding Window Detector

Sliding window is a method for processing a data stream in which a fixed-size window slides over the data, keeps processing the data within the window, and updates the results while the data flow. Inside the window, data can be processed by multiple digital filters. A common digital filter is the **median filter** that substitutes the latest value in the window with the median in the window.

The proposed sliding window detector is formalized in Algorithm 1. Once the extractor outputs the facts of a transaction, it triggers the detector that takes as input the transaction hash and the address of the target contract. The detector also keeps track of the previous transactions in a global variable `factsWindow`. It can be tuned through three parameters: $w1$ is the width of the sliding window, $w2$ is the width of the median filter window, and t is the threshold of absolute deviation of front percentile ranks. The algorithm first reads the fact files as a table and filters the facts: it considers only the target contract's storage slots and removes the facts with $\perp$ values. Then, it computes the differences of the values (i.e., $v_f - v_i$) and updates the `factsWindow` by including the current transaction. For each slot in the current processed facts, the algorithm calculates the percentile rank and median percentile rank of the current transaction in comparison to the transactions stored in the `factsWindow`. If the deviation between the median percentile rank of the current transaction and the average median of the previous transactions is greater than or equal to the threshold t, the `alertLevel` is increased by the deviation. The algorithm returns the `alertLevel`, which is used as an indicator of potential anomalies.

3.4 Alert Reporter

The alert reporter receives the alert levels from the detector and presents them to the practitioner using the methodology. If alert levels are raised for consecutive transactions, the alert reporter groups them and identifies the alert peak, which is the highest alert level for the transaction group. The height of the alert peak represents the level of criticality of the situation during that period.

Algorithm 1 Sliding Window Detector

```
1: global variables: $factsWindow$
2: parameters: $w1, w2, t$
3: inputs: $tx, targetAddress$
4: $facts = getFacts(tx)$
5: $filteredFacts = filterFacts(facts, targetAddress)$
6: $currentFacts = computeDifferences(filteredFacts)$
7: $factsWindow.update(w1, currectFacts)$
8: for all $slot \in currentFacts$ do
9:     $currentFacts[slot].percentileRank = percentileRank(factsWindow, slot)$
10:    $currentFacts[slot].medianPercentileRank = median(factsWindow, slot, w2)$
11:    $averageMedian = averageOfMedian(factsWindow, slot)$
12:    $deviation = abs(currentFacts[slot].medianPercentileRank - averageMedian)$
13:    if $deviation >= t$ then
14:        $alertLevel = alertLevel + deviation$
15:    end if
16: end for
17: return $alertLevel$
```

4 Evaluation

We implemented the proposed methodology in python to evaluate its efficiency and accuracy. The implementation is based on Python 3.9, communicates with Ethereum Mainnet via Ethereum JSON-RPC API `Web3.py` [17]. The implementation and the full result data are available on the GitHub repository `https://github.com/Ressac-No1/EthMainnet_sliding_window_detector`.

4.1 Dataset and Experimental Setup

In order to evaluate the proposed approach, we conducted experiments using a set of DeFi-related smart contracts with known attacks. The dataset used in the experiments, presented in Table 1, consists of smart contracts that had been deployed on Ethereum Mainnet. The contracts were selected based on their relevance to the DeFi ecosystem and the presence of known attacks against them. It is important to note that an attack can contain either single or multiple transactions. The *Start Time* and *End Time* columns indicate the timestamps of the first and last transaction used in the attack, while the *Txns* column represents the number of transactions used in the attack. We included 9 contracts with 11 attacks in our analysis due to the limited resources and time available for the study and the desire to focus on the approach's feasibility. By including these 9 contracts, we were already able to thoroughly analyze and evaluate the proposed approach on a well-defined set of contracts and attacks, which allowed us to demonstrate the approach's effectiveness in a controlled environment while highlighting any limitations and areas for improvement.

The experiments were conducted on a workstation with Intel Core i5-11500H (2.9 - 4.2 GHz) and 8 GB of RAM. The detector parameters $(w1, w2, t)$ were

DeFi Contract	Targeted Contract Address	Attacks	Date	Start Time	End Time	Txns	Single
		1	2021-02-13	04:17:32	07:26:35	7	No
C.R.E.A.M. Finance	0xd06527d5e56a3495252a528c4987003b712860ee	2	2021-08-30	04:03:40	05:44:47	85	No
		3	2021-10-27	13:54:10	13:54:10	1	Yes
TheDAO	0xbb9bc244d798123fde783fcc1c72d3bb8c189413	1	2016-06-17	03:34:48	11:00:23	500	No
Akropolis	0x73fc3038b4cd8ffd07482b92a52ea806505e5748	1	2020-11-12	11:50:41	12:04:37	17	No
Harvest.Finance	0xf0358e8c3cd5fa238a29301d0bea3d63a17bedbe	1	2020-10-26	02:53:58	02:59:22	17	No
Lendf.me	0x0eee3e3828a45f7601d5f54bf49bb01d1a9df5ea	1	2020-04-19	00:58:43	02:12:11	102	No
PAID Network	0x8c8687fc965593dfb2f0b4eaefd55e9d8df348df	1	2021-03-05	17:42:21	18:06:10	29	No
Saddle.Finance	0x5f86558387293b6009d7896a61fcc86c17808d62	1	2022-04-30	08:24:23	08:24:23	1	Yes
Visor Finance	0x3a84ad5d16adbe566baa6b3dafe39db3d5e261e5	1	2021-12-21	14:18:15	14:18:15	1	Yes
Yearn Finance(yDai)	0xacd43e627e64355f1861cec6d3a6688b31a6f952	1	2021-02-04	21:12:40	21:52:15	17	No

Table 1. Dataset: selected contracts and corresponding attacks against them

selected based on a trade-off between computation time and aiming to improve accuracy. We ran the detector with three different configurations of parameters: C1 (500, 50, 0.3), C2 (500, 5, 0.5), C3 (1200, 50, 0.3), and then collected the alert reports for each run and compared them with the location of known attacks.

4.2 Results

For our evaluation, we considered both **efficiency** and **accuracy**.

Results supporting efficiency are shown in Table 2. We considered the following criteria for each configuration: whether or not the detector reports an alert for an attack; the delay of the first alert after the attack starts (D_T_Δ); the first alert level (D_AL); the delay (P_T_Δ) of the closest alert peak after the first alert, and the level (P_AL) of that peak.

		Configuration 1: (500, 50, 0.3)				Configuration 2: (500, 5, 0.5)				Configuration 3: (1200, 50, 0.3)			
DeFi Contract	Atk No.	D_T_Δ	D_AL	P_T_Δ	P_AL	D_T_Δ	D_AL	P_T_Δ	P_AL	D_T_Δ	D_AL	P_T_Δ	P_AL
	1	3d16h	0.651	24min	2.399	3d12h	3.765	0[a]	3.765	3d16h	0.650	24min	2.399
C.R.E.A.M. Finance	2	26min	3.785	1min40s	4.402	Not detected				23min	3.612	57min	4.936
	3	Not detected				Not detected				Not detected			
TheDAO	1	13min	2.371	2min56s	2.394	0	1.947	39s	2.002	13min	2.432	2min54s	2.467
Akropolis	1	13min	0.685	21d12h[b]	0.730	Not detected				13min	0.779	21d12h[b]	0.845
Harvest.Finance	1	1h13min	0.371	29min	0.720	19h	0.924	0[a]	0.924	1h38min	0.712	4min54s	0.721
Lendf.me	1	1h6min	0.436	51min	0.771	24min	1.473	0[c]	1.488	1h6min	0.447	51min	0.851
PAID Network	1	28min	0.304	20min	0.821	30min	0.907	22min	1.311	26min	0.308	1min45s	0.387
Saddle.Finance	1	Not detected				9d6h	1.136	0[a]	1.136	Not detected			
Visor Finance	1	Not detected				59min	1.097	0[a]	1.097	Not detected			
Yearn Finance(yDai)	1	1h52min	0.436	10min	1.174	3d6h	1.472	40s	1.474	41min	0.320	1h21min	1.243

Table 2. Results supporting efficiency

[a]: The detection reached a peak at the first alert
[b]: There were few transactions between the first alert and the peak, although a long delay
[c]: The detection was at a different transaction with the peak, although at the same time

The three configurations were able to detect almost the totality of attacks. Only the third attack on C.R.E.A.M. Finance was not detected. Many factors could contribute to the failure of the detector: this particular attack involved different C.R.E.A.M. Finance contracts (including the one targeted) and was executed using a single transaction. In general, detection of single transaction attacks is less likely to be detected by a broad median-filter window like configurations 1 and 3 than a narrower one like configuration 2. That is because a broad median-filter window may probably smooth out deviated front percentile rank values caused by a single transaction. With regards to the delay, the three configurations behave differently at different attacks, in which the delays vary from a few minutes to several days. That is because our detector is transaction-driven and responds to a potential attack after a number of successive transactions involving the targeted contract, so the detection delay is highly correlated with the transaction frequency shortly after the attacking period.

Results supporting accuracy are shown in Table 3. We considered the following criteria for each configuration: the number of false positives (FP), true positives (TP), and false negatives (FN).

	C1			C2			C3		
DeFi Contract	FP	TP	FN	FP	TP	FN	FP	TP	FN
C.R.E.A.M. Finance	30	2	1	38	1	2	22	2	1
TheDAO	4	1	0	12	1	0	3	1	0
Akropolis	1	1	0	2	0	1	1	1	0
Harvest.Finance	9	1	0	15	1	0	6	1	0
Lendf.me	8	1	0	10	1	0	6	1	0
PAID Network	18	1	0	24	1	0	12	1	0
Saddle.Finance	4	0	1	3	1	0	3	0	1
Visor Finance	1	0	1	0	1	0	1	0	1
Yearn Finance(yDai)	16	1	0	29	1	0	10	1	0

Table 3. Accuracy for the three configurations (C1, C2, C3).

The results clearly show that, in some cases, the detector reported a lot of false positives. It is worth noting that some alerts are not known attacks but may indicate other anomalous situations. For example, investigators on TheDAO attack [18] conducted some white hat attacks against TheDAO contract several days after the real attack happened, which were not adversarial but alerted in our experiments as the detector could not distinguish black and white hat attacks.

4.3 Limitations

Our evaluation results provide valuable insights into the effectiveness of our methodology and will help us identify any areas for improvement. The authors believe this detector can be highly improved and used as a crucial part of a more extensive system for detecting malicious activities in the DeFi ecosystem. The results presented in this study are based on a small set of contracts and attacks, and therefore the findings may not be generalizable to the entire DeFi ecosystem. We envision more forthcoming experiments, including a larger dataset and different configurations. The major limitation of our approach is the occurrence of false positives. The results show that the detector sometimes raises many false alerts, although the false alerts can still be managed by the analysis in a real-time scenario. The authors acknowledge this is not ideal, but it still provides a way to detect attacks before all the funds are drained and enable prompt actions in an emergency. Despite these limitations, the promising results of the study indicate that the detector is robust and can detect the majority of malicious activities with a high recall. This progress highlights the potential for using the detector in real-time to detect fraud and malicious activities in the DeFi ecosystem.

5 Related Work

Several works presented solutions to detect, analyze or anticipate attacks against smart contracts. A vast majority of the presented solutions are based on detecting vulnerabilities via semantic analysis of either the source code or bytecode of a smart contract. These solutions are either based on static analysis such as model checking or symbolic execution (e.g., [6,8,10,12,16,19,22]), or dynamic analysis such as fuzzing (e.g., [3–5,20]). However, they often do not allow developers or investigators to detect attacks on smart contacts as their primary focus is to detect vulnerabilities and not malicious transactions.

Other approaches have been presented to analyze transactions with the focus of detecting attacks and identifying vulnerabilities in smart contracts (e.g., [9,11,15,23])). However, these solutions are not suited for real-time attack detection as they require a significant amount of time to analyze transactions. This is partially due to the fact that they re-execute past transactions with heavy instrumentation.

Finally, there have also been works presented that aim to prevent attacks by eliminating vulnerabilities by applying patches directly to smart contract source code and bytecode (e.g., [7, 21, 24]. While some of these approaches can be used to also detect attacks by replaying past transactions on patched code and comparing differences, they often lack precision and are not sound.

All these aforementioned approaches apply complex and heavy-weight analyses in their frameworks (e.g, symbolic analysis, dynamic taint analysis, etc.). As a result, they are not suitable for real-time detection of smart contract attacks, where a fast reaction time can be crucial. In this work, we present a new attack detection approach that works in real-time while being simple (i.e., analyze abnormal changes in state variables) and fast (i.e. no heavy instrumentation of transaction executions required).

6 Conclusion

Our study aimed to detect DeFi attacks through analyzing the outlier behavior of smart contracts' state variable values. The results of our algorithm showed that it can effectively detect multiple-transaction and single-transaction attacks. However, the current approach still has room for improvement in terms of accuracy, as it generates many false positives. Therefore, future research should focus on improving the accuracy of the detector and expanding its scope to more smart contract platforms and cross-chain attacks. Overall, our study provides a foundation for further investigations into the feasibility of using outlier analysis for DeFi attack detection.

References

1. Ethereum calls table, Infinitable endpoints, Blockchair API documentation, *https : //blockchair.com/api/docs#link_403*
2. Xue, Y., Ma, M., Lin, Y., Sui, Y., Ye, J. and Peng, T.: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In: 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1029–1040. IEEE (2020)
3. Jiang, B., Liu, Y. and Chan, W.K.: Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 259–269. IEEE (2018)
4. Zhang, Q., Wang, Y., Li, J. and Ma, S.: Ethploit: From fuzzing to efficient exploit generation against smart contracts. In: IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 116–126. IEEE (2020)
5. Ashouri, M.: Etherolic: a practical security analyzer for smart contracts. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing, pp. 353–356. 2020
6. Schneidewind, C., Grishchenko, I., Scherer, M. and Maffei, M.: ethor: Practical and provably sound static analysis of ethereum smart contracts. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 621–640. 2020
7. Rodler, M., Li, W., Karame, G.O. and Davi, L.: {EVMPatch}: Timely and automated patching of ethereum smart contracts. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1289–1306. 2021
8. Luu, L., Chu, D.H., Olickel, H., Saxena, P. and Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 254–269. 2016
9. Chen, T., Cao, R., Li, T., Luo, X., Gu, G., Zhang, Y., Liao, Z., Zhu, H., Chen, G., He, Z. and Tang, Y.: SODA: A Generic Online Detection Framework for Smart Contracts. In: NDSS (2020)
10. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F. and Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82. 2018
11. Rodler, M., Li, W., Karame, G.O. and Davi, L., 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. arXiv preprint arXiv:1812.05934 (2018)
12. Stephens, J., Ferles, K., Mariano, B., Lahiri, S. and Dillig, I.: SmartPulse: automated checking of temporal properties in smart contracts. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 555–571. IEEE (2021)
13. Layout of State Variables in Storage, Solidity 0.8.17 documentation, *https : //docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html*
14. Rahimian, R. and Clark, J.: TokenHook: Secure ERC-20 smart contract. arXiv preprint arXiv:2107.02997 (2021)
15. Zhang, M., Zhang, X., Zhang, Y. and Lin, Z.: {TXSPECTOR}: Uncovering attacks in ethereum from transactions. In 29th USENIX Security Symposium (USENIX Security 20), pp. 2775–2792. 2020
16. Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R. and Scholz, B.: Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981 (2018)

17. Web3 API, Web3.py 5.31.3 documentation,
https : //web3py.readthedocs.io/en/v5/web3.main.html
18. A white hat attacker against TheDAO to simulate the attack,
https : //etherscan.io/address/0x2ba9d006c1d72e67a70b5526fc6b4b0c0fd6d334
19. Kalra, S., Goel, S., Dhawan, M. and Sharma, S.: Zeus: analyzing safety of smart contracts. In: NDSS (2018), pp. 1–12.
20. Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y. and Minh, Q.T.: sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 778–788.
21. Nguyen, T.D., Pham, L.H. and Sun, J.: SGUARD: towards fixing vulnerable smart contracts automatically. In: IEEE Symposium on Security and Privacy (SP), pp. 1215–1229. IEEE (2021)
22. Ferreira Torres, C., Schütte J., and State R.: Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference. (2018)
23. Ferreira Torres, C., Iannillo, A. K., Gervais, A., and State, R.: The eye of horus: Spotting and analyzing attacks on ethereum smart contracts. In 25th International Conference in Financial Cryptography and Data Security. (2021)
24. Ferreira Torres, C., Jonker, H., and State, R.: Elysium: Context-Aware Bytecode-Level Patching to Automatically Heal Vulnerable Smart Contracts. In Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses. (2022)