Oracle-Guided Vulnerability Diversity and Exploit Synthesis of Smart Contracts Using LLMs

Mojtaba Eshghie
KTH Royal Institute of Technology
Stockholm, Sweden
eshghie@kth.se

Cyrille Artho
KTH Royal Institute of Technology
Stockholm, Sweden
artho@kth.se

ABSTRACT

Many smart contracts are prone to exploits, which has given rise to analysis tools that try to detect and fix vulnerabilities. Such analysis tools are often trained and evaluated on limited data sets, which has the following drawbacks: 1. The ground truth is often based on the verdict of related tools rather than an actual verification result; 2. Data sets focus on low-level vulnerabilities like reentrancy and overflow; 3. Data sets lack concrete exploit examples. To address these shortcomings, we introduce XploGen, which uses a model-based oracle specification of the business logic of the smart contracts to synthesize valid exploits using LLMs. Our experiments, involving 104 synthesized vulnerability-exploit pairs, demonstrated a 57% success rate in exploiting targeted aspects of the contract. They achieved exploit efficiency with an average of only 3.5 transactions per exploit, highlighting the effectiveness of our methodology.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*; • **Software and its engineering** → **Software verification and validation**.

KEYWORDS

Exploit Synthesis, Smart Contract, Vulnerability, LLM, Large Language Models

ACM Reference Format:
Mojtaba Eshghie and Cyrille Artho. 2024. Oracle-Guided Vulnerability Diversity and Exploit Synthesis of Smart Contracts Using LLMs. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3691620.3695292

1 INTRODUCTION

Smart contracts are self-executing programs that encode traditional contract terms enabling decentralized applications. Despite their potential, they are vulnerable to exploitation, especially through business logic flaws, where the contract deviates from intended operations, leading to significant financial losses. Exploits of business logic vulnerabilities in smart contracts have often led to significant losses. The DeFiHackLabs repository lists daily updated attacks and their reproducible exploits, recording 70 successful attacks in 2024 [28]. Of these, 17 (about 24 %) are due to *business logic flaws*, resulting in losses of about 52 million USD losses (24% of the total losses as of May 29, 2024), underscoring the severity of such flaws.

Previous research has largely overlooked business-logic vulnerabilities, focusing instead on well-known issues like reentrancy and integer overflow [10, 29]. This is because the business logic of a smart contract is application-specific. Identifying business logic flaws requires understanding the contract's intended behavior. These flaws are not easily recognizable as they deviate from the expected program behavior and do not follow traditional patterns, complicating their detection by (especially static) analysis tools.

Automated detection of logic flaws relies on a formal, machine-readable description of the contract's intended behavior. In this work, we demonstrate that such a formal model can be utilized not only by specialized tools but also by large language models (LLMs). By incorporating the model into our prompting strategy, we use the formal specification to generate precise, context-aware exploits.

Realistic business logic vulnerabilities are rare in existing data sets, which tend to focus on simple low-level implementation flaws [3, 5, 11, 12, 35, 36]. To remedy this, we use LLMs to diversify existing smart contract code to weaken it specifically in a way that violates our formal specification.

The generated vulnerability-exploit pairs[1] contribute to the state of the art in two ways:

(1) Synthesized exploits that successfully violate the specifications provide a witness of how the implementation deviates from the expected behavior. This witness not only helps contract developers, but also helps training and evaluating existing automatic analysis, synthesis, and repair tools for smart contracts [7, 9, 13, 14, 16, 22, 33].
(2) We leverage Dynamic Condition Response (DCR) graphs as a business-logic specification. DCR graphs embody complex dependencies and constraints in an accessible, graphical notation. We apply them both statically (guiding an LLM in producing vulnerable mutants and diversifying exploits) and dynamically (as a runtime oracle to confirm whether an exploit is valid) [6].

2 METHODOLOGY

Our methodology (see Fig. 1) consists of an LLM agent synthesizing vulnerabilities and diversifying exploits based on a context enriched with formal business logic specification of the contract (prompts in Fig. 3 and Fig. 6). We use the

[1] https://github.com/mojtaba-eshghie/XploGen

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695292

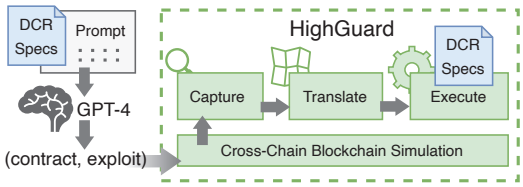

Figure 1: Experiment setup

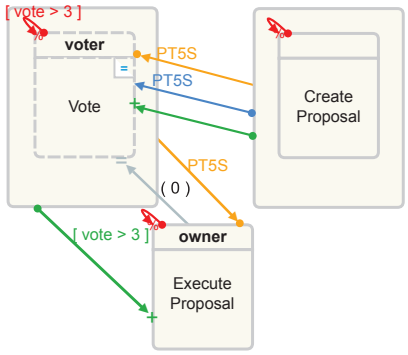

Figure 2: Governance contract DCR model.

Table 1: Effectiveness of injected vulnerabilities and synthesized exploits.

Contract	Synthesized contract variants	Compilable contract variants	Exploitable pairs
Governance	25	15	15
Escrow	4	4	2
MultiStageAuction	25	25	13
PrizeDistribution	25	23	7
ProductOrder	25	24	15
Total	104	91	52

experimental setup in Fig. 1 to diversify the vulnerabilities, generate the exploits, and test them.

Our setup benefits from HighGuard, a monitoring tool that monitors the contract with respect to its DCR specifications [6, 8]. DCR is a declarative modeling notation primarily used to model business processes [15]. DCR has been established as a suitable formalism for business logic of the smart contracts capturing various aspects such as access control, time-based properties, as well as partial ordering of interactions with the contract [8]. As Fig. 1 shows, HighGuard (monitor) uses the same DCR specification used in our prompting for malicious transaction detection.

Fig. 2 shows a simplified version of DCR model for the Governance contract. This model captures aspects related to the business logic of the contract that are independent from the platform-specific implementation details, such as time and temporal properties and role-based access control.

3 GENERATION OF VULNERABILITIES AND EXPLOITS

We modeled five contracts from various application areas using DCR graphs (see Table 1), incorporating common business logic patterns found in smart contract development [8]. The DCR models were kept constant, while vulnerabilities and target exploits were synthesized 25 times per contract using GPT-4 with the one-shot prompt in Fig. 3.

```
I have modeled a smart contract using DCR graphs.
You are provided: semantics of DCR graphs, the contract
in Solidity, its DCR model, the exploit that abuses a
vulnerability in the contract.

DCR semantics summary in natural language

```solidity
// Complete code of the contract
```

```xml
// Summarized DCR model
```

```javascript
// Exploit function source code
```

Task: To instrument the contract to change the
vulnerability and create another business-logic
vulnerability. Only the new contract and exploit are to be
returned, with no further explanation. The exploit should
try to break the properties defined in the DCR model.
```

Figure 3: Oracle-guided prompt used to synthesize vulnerable contract and exploits.

It is crucial that the contracts created in this step implement the same business logic as the one from the seed contract from which the automatic instrumentation began (oracle-guided prompt in Fig. 3). Therefore, we need variations of the contract that are modified *minimally* to introduce a possible exploit on the contract.

Using the prompt in Fig. 3 resulted in variants of the original smart contract that are vulnerable to business logic flaw vulnerabilities as well as their concrete exploit (JavaScript).

The generated exploits lead to failures in the business logic of the contracts. The most severe exploits resemble successful attacks and enable an unauthorized execution of functions, bypassing key conditional checks [28]. As an example of an injected vulnerability, a *require* statement in function *vote* of the *Governance* contract is removed, which was used to check the possibility of voting based on time. The respective exploit function synthesized by the LLM tries to exploit this vulnerability by sending transactions to function *vote* outside of expected times. Another injected vulnerability allows for the execution of proposals in the Governance contract without receiving enough votes, and the respective exploit tries to trigger this vulnerability by initiating transactions from different accounts.

The synthesis goal for each of the five contracts in Table 1 is to obtain 25 variants with injected vulnerabilities. In case of the Escrow contract, only 4 of the generated variants were sensible without introducing new contracts or drastically changing or removal of existing core contract functionality.

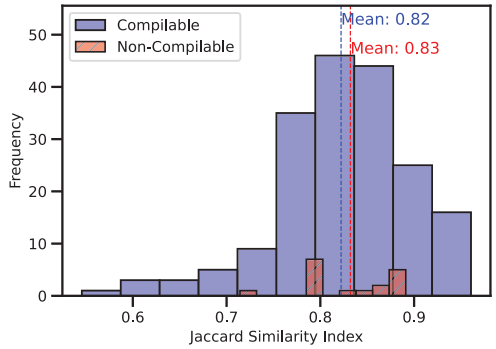

Figure 4: Distribution of similarity by compilability

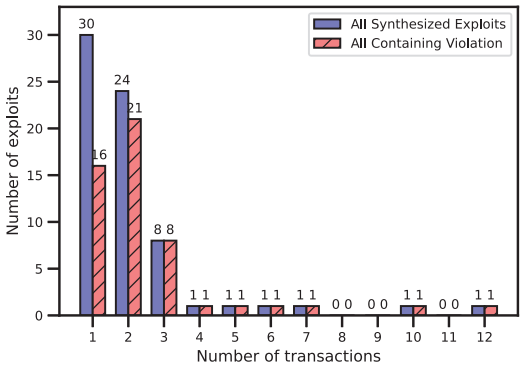

Figure 5: Number of successful transactions monitored involved in synthesized contract exploits

This was determined by reviewing all output contracts of the LLM before using them in our experiment pipeline (Fig. 1).

Out of 104 synthesized contracts, 91 were compilable. These contracts differ from the original by adding or removing code chunks, raising the question of whether compilability is related to the degree of difference from the original code. We measure this difference using the Jaccard index, which quantifies similarity between two sets of data [21, 25].

Fig. 4 shows the distribution of code similarity between their synthesized variants and corresponding original contract grouped into *compilable* and *non-compilable* bars.

The Jaccard similarity indices for compilable and non-compilable contracts are very close (0.82 vs. 0.84), with considerable overlap between their distributions. This suggests that Jaccard similarity does not effectively distinguish between the original contracts and their vulnerable counterparts in terms of compilability.

To clarify the results, an *exploitable pair* consists of a vulnerable contract and its corresponding exploit—a sequence of transactions designed to alter the contract's state in a way that violates its business logic. The exploit must successfully execute at least one transaction that breaches defined contract properties. We refer to the JavaScript function that initiates this sequence as the exploit.

Table 2: Vulnerability-exploit synthesis using DCR-based oracle (Section 3) against exploit diversification with code comments (Section 4)

Experiment	With Executable Transactions	Exploitable pairs	Avg. # Transactions per Exploit
Synthesis (§3)	12	7	3.5
Diversification (§4)	10	6	20.1

The 91 generated exploits initially involve a total of 492 transactions. However, during runtime, only 146 of these transactions successfully execute.

Fig. 5 shows the transaction distribution per exploit. Based on this plot, the majority of all executable exploits have only 1, 2, or 3 transactions, depending on the contract's logic.

Table 1 summarizes the results of deploying injected contracts and executing exploits under HighGuard's monitoring. Out of 91 compilable contracts, 52 were exploited successfully. However, 23 exploits failed to execute any successful transactions, and many others could not complete the full exploit sequence. Only 15 of the 52 exploitable pairs fully executed their exploit sequences.

4 EXPLOIT DIVERSIFICATION

To show the effect of using a machine-readable specification in the prompt, we conducted another experiment with the prompt in Fig. 6. We manually inject multiple vulnerabilities into the *PrizeDistribution* contract to make sure there are multiple ways to exploit the contract.

````
Below, you have a smart contract that is vulnerable
to business logic flaw vulnerability. After that, you
are provided the JavaScript exploit that abuses the
vulnerability in my contract is given.

```solidity
// Complete code of the contract
```

```javascript
// Exploit function source code
```

Task: To create other exploits of the same type. Do NOT try
to exploit generic vulnerability types such as reentrancy,
etc. I only need the new exploit function.
````

Figure 6: The prompt for exploit diversification (without machine-readable DCR oracle)

We use the comments in the example contract with natural language information of the vulnerability in the source code of *PrizeDistribution*. As the prompt does not contain any further information about the business logic of the contract, the aforementioned comment is essential s. t. the synthesized exploits target business logic vulnerabilities.

We executed the exploits against isolated *PrizeDistribution* deployments, monitored by HighGuard. Table 2 compares these results with the DCR-based oracle experiment (Section 3). As the table shows, exploit diversification with

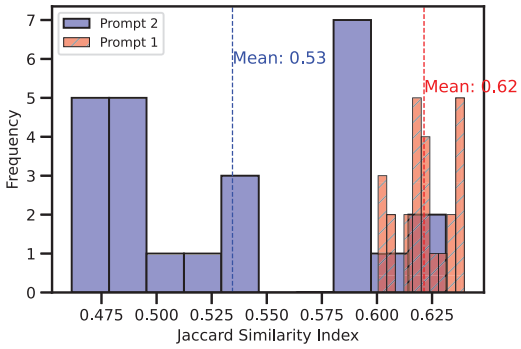

Figure 7: Distribution of similarities between synthesized exploits using prompt in Fig. 3 and prompt in Fig. 6 for PrizeDistribution Contract.

comment-based vulnerability information does not surpass the oracle-guided generation.

Although the exploits generated by the oracle-guided prompt (Fig. 3) were not significantly more effective (exploitable column in Table 2), they were significantly more efficient in directly exploiting the contract with a minimal number of transactions.

Another way to compare the synthesized exploits is to investigate their similarity with the original example exploits that were provided in prompts to the model. The plot in Fig. 7 demonstrates the distribution of similarity for both experiments. Interestingly, the exploits of the machine-readable oracle (Section 3) prompt are closer to the original example overall (mean 0.62). Exploits generated from our prompt in Fig. 3 also have less variation in their distance as opposed to the ones resulted from the prompt in Fig. 6.

5 RELATED WORKS

Exploit Synthesis. Previous works on exploit synthesis, such as TestBreeder, EthPloit, ExGen, TeEther, and ContraMaster, primarily focus on generating exploits for traditional vulnerabilities like reentrancy, integer overflow, and unchecked transfer values [17–19, 30, 34]. TestBreeder uses automatic test case generation to evaluate smart contract analysis tools, gradually increasing the complexity of test cases to uncover analyzers' weak points [18]. EthPloit employs fuzzing and static taint analysis, while ExGen translates contracts into an intermediate representation for symbolic attack contract generation [34]. TeEther uses control flow graphs and symbolic execution [19]. ContraMaster mutates transaction sequences to create exploits, with a focus on temporal order and semantic mismatches [30].

Compared to XploGen, the aforementioned exploit generation tools are deficient in several aspects: 1) They do not support smart contracts written in more recent versions of Solidity [19]. Our experiments in Section 3 conducted on smart contracts with Solidity language ranging from 0.6.x to 0.8.0 demonstrate effectiveness of XploGen across versions. 2) They lack reproducibility [18]. 3) The oracle for exploit generation is limited in expressiveness, informally defined, or pertains to low-level vulnerabilities [18, 31, 34].

LLM-Based Works. PropertyGPT, GPTScan, and SmartInv all leverage LLMs for detection of smart contract vulnerabilities and generation of properties [20, 27]. PropertyGPT addresses the *specification* bottleneck in formal verification by embedding human-written properties into a vector database, which allows the model to retrieve and adapt these properties for new contracts through in-context learning [20, 26]. GPTScan focuses specifically on business logic vulnerabilities, using a breakdown of logic vulnerabilities into specific scenarios to enable LLMs to identify potential issues, which are then validated by static analysis to reduce false positives [27]. SmartInv integrates source code with natural language specifications of the contracts. It utilizes a Tier of Thought (ToT) prompting strategy to fine-tune models in stages, localizing vulnerabilities [32]. Although these works leverage LLMs, they fall short of XploGenin providing concrete exploits as evidence of discovered vulnerabilities.

Exploit Databases. Many vulnerability and attack listings offer detailed descriptions without providing concrete, executable exploits [1, 4, 24]. In contrast, some repositories include reproducible exploits for successful attacks, yet they cover only a subset of real-world attacks, limiting their diversity and comprehensiveness [23, 28]. Our approach addresses this gap by generating a diverse set of vulnerability-exploit pairs that are both executable and specifically tailored to demonstrate business logic flaws.

6 CONCLUSION

We introduced a systematic approach to vulnerability injection and exploit synthesis for smart contracts using LLMs. By incorporating machine-readable business logic specifications in the prompt, we generated accurate, context-aware vulnerability-exploit pairs. In our experiments with 104 synthesized pairs, 91 were compilable, and 52 successfully violated the contract's intended business logic, showcasing the effectiveness of our method.

The experiments highlighted that exploits generated with business logic context are efficient, with an average of about 3.5 transactions per exploit compared to about 20 transactions per exploit without the context, which underscores the significance of incorporating formalized business logic.

7 FUTURE WORK

Future research should explore if XploGen's effectiveness varies with the type of property used. Additionally, a manual review of failed exploits will clarify whether the target contracts lack logic vulnerabilities or if the issue lies in the exploit's implementation.

While our method currently synthesizes JavaScript exploits, directly synthesizing attacker contracts in Solidity using platforms like Foundry [2] could provide access to low-level EVM features and enhance exploit development.

Our experiments involve contracts averaging around 100 lines of code. Larger contracts may affect the quality of injected vulnerabilities and exploit effectiveness [21]. Extending our approach to cross-platform settings could further expand its scope.

REFERENCES

[1] 2024. Comprehensive List of DeFi Hacks & Exploits. https://chainsec.io/defi-hacks/

[2] 2024. Foundry-Rs/Foundry. Foundry. https://github.com/foundry-rs/foundry

[3] 2024. Smartbugs/Smartbugs-Curated. SmartBugs.

[4] 2024. Yearn-Security/Disclosures at Master · Yearn/Yearn-Security. https://github.com/yearn/yearn-security/tree/master/disclosures

[5] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Ben Livshits. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. https://doi.org/10.1145/3597503.3623302 arXiv:2304.02981 [cs]

[6] Mojtaba Eshghie. 2024. Mojtaba-Eshghie/HighGuard. https://github.com/mojtaba-eshghie/HighGuard

[7] M Eshghie, W Ahrendt, C Artho, TT Hildebrandt, and G Schneider. 2023. CLawK: Monitoring Business Processes in Smart Contracts (2023). *DOI: https://doi. org/10.48550/arXiv* 2305 (2023).

[8] Mojtaba Eshghie, Wolfgang Ahrendt, Cyrille Artho, Thomas Troels Hildebrandt, and Gerardo Schneider. 2023. Capturing Smart Contract Design with DCR Graphs. https://doi.org/10.48550/arXiv.2305.04581 arXiv:2305.04581 [cs].

[9] Mojtaba Eshghie, Cyrille Artho, and Dilian Gurov. 2021. Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning. In *Evaluation and Assessment in Software Engineering (EASE 2021)*. Association for Computing Machinery, New York, NY, USA, 305–312. https://doi.org/10.1145/3463274.3463348

[10] 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.

[11] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. https://doi.org/10.48550/arXiv.2007.04771 arXiv:2007.04771 [cs]

[12] Asem Ghaleb and Karthik Pattabiraman. 2020. How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 415–427. https://doi.org/10.1145/3395363.3397385 arXiv:2005.11613 [cs]

[13] 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.

[14] Hanyang Guo, Yingye Chen, Xiangping Chen, Yuan Huang, and Zibin Zheng. 2024. Smart Contract Code Repair Recommendation Based on Reinforcement Learning and Multi-metric Optimization. *ACM Transactions on Software Engineering and Methodology* 33, 4 (April 2024), 106:1–106:31. https://doi.org/10.1145/3637229

[15] Thomas T. Hildebrandt, Håkon Normann, Morten Marquard, Søren Debois, and Tijs Slaats. 2022. Decision Modelling in Timed Dynamic Condition Response Graphs with Data. In *Business Process Management Workshops*. Springer, Cham, 362–374.

[16] Hai Jin, Zeli Wang, Ming Wen, Weiqi Dai, Yu Zhu, and Deqing Zou. 2021. Aroc: An automatic repair framework for on-chain smart contracts. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4611–4629.

[17] Ling Jin, Yinzhi Cao, Yan Chen, Di Zhang, and Simone Campanoni. 2023. ExGen: Cross-platform, Automated Exploit Generation for Smart Contract Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 20, 1 (Jan. 2023), 650–664. https://doi.org/10.1109/TDSC.2022.3141396

[18] Ki Byung Kim and Jonghyup Lee. 2020. Automated Generation of Test Cases for Smart Contract Security Analyzers. *IEEE Access* 8 (2020), 209377–209392. https://doi.org/10.1109/ACCESS.2020.3039990

[19] 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.

[20] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2024. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. arXiv:2405.02580 [cs]

[21] Gabriele Morello, Mojtaba Eshghie, Sofia Bobadilla, and Martin Monperrus. 2024. DISL: Fueling Research with A Large Dataset of Solidity Smart Contracts. https://doi.org/10.48550/arXiv.2403.16861 arXiv:2403.16861 [cs]

[22] 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.

[23] sallywang147. 2024. Sallywang147/attackDB. https://github.com/sallywang147/attackDB

[24] sayan. 2024. Sayan011/Immunefi-bug-bounty-writeups-list. https://github.com/sayan011/Immunefi-bug-bounty-writeups-list

[25] André Storhaug. 2024. Andstor/Verified-Smart-Contracts. https://github.com/andstor/verified-smart-contracts

[26] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. 2024. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. https://doi.org/10.48550/arXiv.2401.16185 arXiv:2401.16185 [cs]

[27] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2023. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. https://doi.org/10.48550/arXiv.2308.03314 arXiv:2308.03314 [cs]

[28] SunWeb3Sec. 2023. DeFi Hacks Reproduce - Foundry. https://github.com/SunWeb3Sec/DeFiHackLabs

[29] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference*. 664–676.

[30] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2022. Oracle-Supported Dynamic Exploit Generation for Smart Contracts. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (May 2022), 1795–1809. https://doi.org/10.1109/TDSC.2020.3037332

[31] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2022. Oracle-Supported Dynamic Exploit Generation for Smart Contracts. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (May 2022), 1795–1809. https://doi.org/10.1109/TDSC.2020.3037332 Conference Name: IEEE Transactions on Dependable and Secure Computing.

[32] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. 2024. SMARTINV: Multimodal Learning for Smart Contract Invariant Inference. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 125–125. https://doi.org/10.1109/SP54263.2024.00126

[33] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. {TXSPECTOR}: Uncovering Attacks in Ethereum from Transactions. In *29th USENIX Security Symposium (USENIX Security 20)*. 2775–2792.

[34] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. 2020. EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 116–126. https://doi.org/10.1109/SANER48275.2020.9054822

[35] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. 2023. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. https://doi.org/10.48550/arXiv.2305.08456 arXiv:2305.08456 [cs]

[36] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. SoK: Decentralized Finance (DeFi) Attacks. https://doi.org/10.48550/arXiv.2208.13035 arXiv:2208.13035 [cs].

Accepted 3 August 2024