Repairing DeFi Vulnerabilities: Benchmarking LLMs with Executable Solidity Exploits

Lucas Bastos Germano
lucas.germano@ime.eb.br
Military Institute of Engineering
Rio de Janeiro, RJ, Brazil

Julio Cesar Duarte
duarte@ime.eb.br
Military Institute of Engineering
Rio de Janeiro, RJ, Brazil

ABSTRACT

Decentralized finance protocols are frequently exploited, creating a demand for fast and reliable repair of vulnerable smart contracts and validation that reflects runtime security. Large language models are an emerging source of patches, yet many evaluations rely on manual checks or self-assessment, which cannot confirm whether attacker profit is actually prevented. We introduce an executable benchmark that replays verified real-world exploits against patched Solidity contracts under a resilient protocol that permits alternate attack paths and controlled state variation. Our framework compiles candidate patches, deploys them on a forked chain, and tests whether the exploit still yields profit. The benchmark covers six test cases drawn from reproducible incidents and is released as open-source. Among the nine evaluated models, GPT-5, GPT-4.1, and Claude Opus 4.1 performed the best, mitigating four of six test cases. Microsoft Phi-4 was the most reliable open-source model, mitigating two of six exploits and producing compilable patches for the remaining cases. No model mitigated the H2O case once resilient checks were enabled, while a simpler access control flaw, BTNFT, was often repaired with minimal edits. Grounding validation in executable exploit replay provides a precise and scalable method to measure whether proposed repairs harden contracts at runtime.

KEYWORDS

large language models, vulnerability repair, defi, automated testing

1 INTRODUCTION

Decentralized finance (DeFi) has transformed financial services by enabling trustless, programmable transactions via smart contracts. However, this advancement poses significant security risks, as DeFi protocols have suffered several exploits in recent years, resulting in the loss of billions of dollars. In the first half of 2025 alone, over $2.17 billion in cryptocurrency was stolen, surpassing the total for all 2024 [9]. A notable incident occurred in July 2025, when the GMX protocol on the Arbitrum network suffered a reentrancy attack that exploited a design flaw in its liquidity pool price calculations, causing losses estimated at $40–42 million [13].

In response to these recurring threats, researchers have increasingly turned to artificial intelligence for solutions, as Large Language Models (LLMs) now show remarkable promise for improving security [6]. Recent research demonstrates that advanced LLMs can automatically perform a range of vulnerability-related tasks, including detecting vulnerabilities in source code [4, 14, 16], explaining the underlying causes and potential impacts of these vulnerabilities [3], and generating candidate repairs or patches, often outperforming traditional static analysis tools [5, 18]. In the domain of smart contracts, frameworks such as ContractTinker [18] have achieved high accuracy in vulnerability localization and repair.

Despite recent progress in LLM-based vulnerability repair, evaluations typically rely on manual review or assessments produced by other LLMs. While these approaches can provide initial insights, they are inherently limited and prone to subjectivity. Manual validation is time-consuming and not scalable, whereas LLM-based evaluations may overlook subtle flaws or incorrectly assume that a vulnerability has been resolved based on superficial syntactic patterns. As a result, a contract may appear correctly patched while remaining exploitable in practice.

To address these limitations, our work proposes a rigorous, automated evaluation strategy. We build on the reproducible exploit dataset provided by DeFiHackLabs [15] and systematically replay each real exploit against its patched contracts. To improve robustness, the original exploit scripts are adapted to allow more flexible attack paths, ensuring that vulnerabilities are still detected even when a repair partially mitigates the issue. This process verifies not only syntactic validity and successful compilation, but also confirms whether the exploit is effectively mitigated. By simulating the original attack scenarios, we ensure repairs are both functionally sound and security-effective, an aspect often overlooked in prior work.

Our contributions are fourfold. First, we introduce a reproducible benchmark based on real-world DeFi exploits, comprising verified vulnerable contracts and associated attack scripts customized for automated evaluation. Second, we propose an executable validation framework that replays original exploits against repaired contracts, assessing both compilation success and whether the exploit remains viable. Third, we conduct a comparative study of closed- and open-source LLMs, analyzing their effectiveness in repairing vulnerabilities under this evaluation framework. Finally, we release our framework and benchmark as open source, enabling reproducibility, transparency, and future extensions by the research community[1].

By grounding vulnerability repair in concrete exploit replays, our benchmark establishes a new standard for evaluating whether LLM-based security patches effectively prevent losses in deployed smart contracts. We instantiate the benchmark with 6 real-world DeFi incidents, representing approximately $1.8 million in historical losses. Among closed-source models, GPT-5, GPT-4.1, and Claude

[1]https://github.com/lucasg1/reproducible-solidity-repair-benchmark

In: I Workshop Brasileiro de Sistemas Web3 (BrWeb3 2025). Anais Estendidos do XXXI Simpósio Brasileiro de Sistemas Multimídia e Web (BrWeb3'2025). Rio de Janeiro/RJ, Brasil. Porto Alegre: Brazilian Computer Society, 2025.

ISSN 2596-1683

Table 1: Comparison of smart contract repair approaches

Work	Evaluated LLMs	Replay Validation	Scope
Zhang et al. [21]	GPT-4, GPT-3.5, Mistral-7B, Llama3-8B, Llama3.2-11B	✓	Access Control
Wang et al. [18]	GPT-4, GPT-3.5	✗	General
Jain et al. [5]	GPT-3.5-Turbo, Llama2-7B	✗	General
Napoli and Gatteschi [10]	GPT-3.5-Turbo	✗	General
This work	**GPT-5, GPT-4.1, GPT-4o, Claude Opus 4.1, Claude Sonnet 3.7, Phi-4 14B, Qwen2.5 Coder32B, Qwen3 32B, DeepSeek Coder 33B**	✓	**General**

Opus 4.1 performed similarly, successfully mitigating 4 out of 6 exploits. For open-source models, Phi-4 and Qwen2.5 Coder 32B were the top performers, mitigating 2 exploits. Notably, Phi-4 exhibited strong consistency in code generation, with all outputs compiling successfully, even when the vulnerability was not fully resolved.

The rest of the paper is organized as follows: Section 2 discusses existing approaches to vulnerability repair in smart contracts. Section 3 details the proposed repair framework, while Section 4 presents our experimental setup and the comparative performance of the evaluated models. Finally, Section 5 summarizes our findings and outlines future directions.

2 RELATED WORK

To contextualize our executable replay-based repair framework, we contrast it with four recent LLM-powered smart contract repair approaches: the Two Timin' pipeline proposed by Jain et al. [5], the ContractTinker tool from Wang et al. [18], the ACFix framework of Zhang et al. [21], and the ChatGPT-based repair study conducted by Napoli and Gatteschi [10].

Jain et al. [5], Wang et al. [18] demonstrated that combining static analysis with advanced prompting can eliminate many vulnerabilities at the source-code level. However, they did not perform exploit-level validation.

Zhang et al. [21] introduced a novel framework, ACFix, that mines common role-based access control (RBAC) practices from over 344K smart contracts to guide LLMs in repairing access control vulnerabilities. They used replay validation from the same dataset, but only for access control cases. ACFix achieved strong correctness on such cases, validated through static and semantic checks with multi-agent debate, while not addressing general vulnerabilities.

Napoli and Gatteschi [10] investigated the use of ChatGPT for repairing Solidity smart contracts, focusing on its capacity to produce syntactically correct and compilable patches. Their evaluation is based solely on static correctness checks, without incorporating runtime replay validation or addressing a broad range of vulnerability types.

Our approach complements and extends these prior efforts by introducing the first correction validation pipeline that replays the original exploit on the patched contract, ensuring that vulnerabilities are removed at execution time. This replay-based validation applies to any vulnerability with a known exploit, whereas ACFix [21] restricts its replay checks to access control cases. As shown in Table 1, the other compared methods rely solely on static or semantic analysis, which cannot guarantee exploit resilience in practice.

In addition, our work evaluates the widest range of LLMs to date for this task, including the largest open-source models across related works (up to 33B parameters) and the only evaluation involving a state-of-the-art model outside the OpenAI ecosystem. We also address the reproducibility gap noted in prior literature [7] by making our benchmark publicly available, enabling fair comparison and future research.

3 METHODOLOGY

This section describes the design and implementation of our vulnerability repair framework, which automates the repair and validation of vulnerabilities in DeFi smart contracts using LLMs. It supports both local models and API-based models from OpenAI and Claude.

3.1 Framework Overview

The proposed framework takes as input vulnerable smart contracts from the DeFiHackLabs dataset [15] along with their vulnerability descriptions. It automates the repair process and evaluates whether the generated patches eliminate the vulnerability by replaying the original exploits. This workflow, illustrated in Figure 1, consists of the following key steps:

1. **Contract Source Retrieval**: obtains the verified Solidity source code of the contract from the DeFiHackLabs dataset.
2. **Vulnerable Code Extraction**: parses that source code to isolate the main contract, excluding secondary contracts (e.g., ERC20, Ownable, SafeMath).
3. **LLM Querying**: constructs a repair prompt combining the extracted fragment and its vulnerability description, then invokes the selected LLM.
4. **Repair Contract**: inserts the LLM-provided patch into the full contract to produce a repaired version.
5. **Bytecode Swap**: compiles the patched contract and replaces the bytecode at the original deployed address.
6. **Exploit Reproduction on Patched Code**: re-runs the exploit script against the patched contract and records whether the exploit still succeeds.

3.2 LLMs Selection

Due to the fast-paced growth and change in this area, we selected LLMs based on two criteria: (1) hardware availability for local models (detailed in Section 4), specifically those fitting within 64 GB VRAM; and (2) performance, with models chosen according to their rankings on established benchmarks and leaderboards [1, 2, 12, 19, 20]. Our framework supports API-based models from OpenAI and

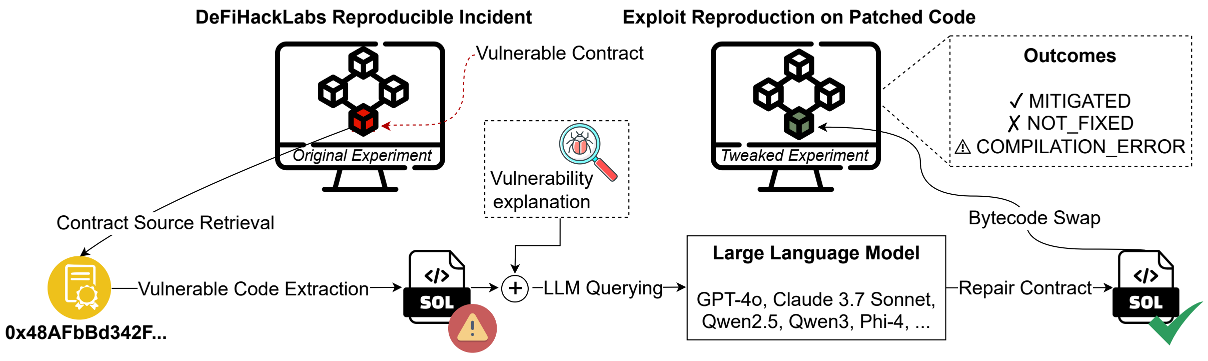
DeFiHackLabs Reproducible Incident
Exploit Reproduction on Patched Code
Vulnerable Contract
Original Experiment
Tweaked Experiment
Outcomes
✓ MITIGATED
✗ NOT_FIXED
⚠ COMPILATION_ERROR
Vulnerability explanation
Contract Source Retrieval
Bytecode Swap
Vulnerable Code Extraction
LLM Querying
Large Language Model
GPT-4o, Claude 3.7 Sonnet, Qwen2.5, Qwen3, Phi-4, ...
Repair Contract
SOL
0x48AFbBd342F...

Figure 1: Methodology employed in the experiment.

Claude, and any open-source model from the HuggingFace Hub. Based on the two criteria described, the following models were selected:

- API-based models: GPT-5 [2], GPT-4.1, GPT-4o, Claude Opus 4.1, and Claude Sonnet 3.7
- Local models: Phi-4 14B, Qwen3 32B, Qwen2.5 Coder 32B Instruct, and Deepseek Coder 33B Instruct

3.3 Repair Validation

After the repaired contract is generated, the framework replaces the original implementation with the patched version. It then runs an automated test script that replays the original exploit using Foundry [11], an Ethereum development toolkit that enables local Ethereum Virtual Machine (EVM) testing for realistic exploit replay. The output is categorized as:

✓ MITIGATED – Exploit successfully mitigated;
✗ NOT_FIXED – Exploit still succeeds; and
⚠ COMPILATION_ERROR – Patched code failed to compile

If the result is `NOT_FIXED`, the exploit remains fully viable, indicating that the generated repair failed to address the underlying vulnerability. Conversely, a `MITIGATED` result guarantees that the exploit is no longer applicable. This outcome may indicate that the vulnerability was completely fixed, or that exploiting it would now require a redesign of the original attack strategy. In this work, the exploit scripts from the DeFiHackLabs dataset [15] were tweaked to allow greater flexibility in executing the attacks. The modified experiments are no longer constrained to the original exploit's exact sequence of actions, making them more robust in detecting vulnerabilities even when the patch only partially mitigates the issue. An example of a tweaked experiment explanation and testing is provided in Subsection 4.2.

4 RESULTS

We evaluate nine large language models across six real-world DeFi incidents by replaying the original exploit against each patched smart contract. This setup yields 54 evaluations, enabling comparisons by model family (closed- vs. open-source), exploitation difficulty, and frequent error modes observed during automated testing. This section reports aggregated results and discussions.

Experiments were conducted on a virtual machine with four NVIDIA A16 GPUs (16 GB each, 64 GB total VRAM), 32 GB RAM, and an 8-core CPU running at 2 GHz. Local models with 32-33 billion parameters were loaded using 4-bit quantization, while Phi-4 was executed in FP16.

4.1 Repair Evaluation Results

Table 2 summarizes repair outcomes based on the criteria in Subsection 3.3. Closed-source models achieved 17 mitigations out of 30 evaluations (56.7%) with only a single compilation failure, whereas open-source models achieved 5 mitigations out of 24 evaluations (20.8%) but encountered 12 compilation failures. GPT-5, GPT-4.1, and Claude Opus 4.1 led overall with 4 mitigations out of 6 cases. Notably, no model was able to mitigate `H2O` after adapting the experiment to allow greater flexibility.

Table 2: Repair results per model across DeFi vulnerabilities.

Model	BBX	BTNFT	DCF	H2O	IPC	SR
GPT-5	✓	✓	✓	✗	✗	✓
GPT-4.1	✓	✓	✗	✗	✓	✓
GPT-4o	✓	✓	✗	✗	✗	⚠
Claude Opus 4.1	✓	✓	✗	✗	✓	✓
Claude Sonnet 3.7	✓	✓	✗	✗	✗	✓
Phi-4 14B	✓	✗	✗	✗	✗	✓
Qwen2.5 32B	✓	⚠	✗	✗	⚠	✓
Qwen3 32B	⚠	⚠	⚠	⚠	✗	⚠
DeepSeek 33B	⚠	⚠	⚠	⚠	⚠	✓

Table Legend: ✓ Exploit mitigated; ✗ Failed repair; ⚠ Compilation failed.

Per-incident difficulty aligns with domain expectations. Simple logic errors, such as `BBX` and `SR`, were mitigated by 7 of 9 models, suggesting that vulnerabilities with direct attack paths are accessible to be fixed with current LLMs. `BTNFT`, a straightforward access control flaw, behaved as expected: most models successfully repaired it with minimal edits, while those prone to compilation instability failed mainly at the build step rather than at runtime validation. `IPC` proved more challenging, with just 2 successful

[2] Although it does not appear in any of the referenced benchmarks due to its recent release, it is expected to provide improvements in reasoning and code generation over prior GPT-4 models

mitigations compared to 5 failed repairs and 2 compilation failures. `DCF` showed high difficulty with only one mitigation, as it involves transfer-triggered side effects in automated market makers. `H2O` was not mitigated by any model after adding flexibility to the replay, reflecting its high repair complexity; this case will be described in detail in Subsection 4.2.

Compilation stability differentiates families. Closed-source models were consistent, with only one compilation failure in 30 evaluations. Open-source models produced 12 failures in 24 evaluations, concentrated in `Qwen3 32B` and `DeepSeek 33B`. Among them, `Phi-4 14B` and `Qwen2.5 32B` were most reliable, each producing 2 mitigations, with `Phi-4 14B` compiling in all cases.

Error modes were repeatable. Frequent issues included partial patches that left helper functions undefined, inconsistent interface or import changes that broke builds, and empty or truncated generations that caused silent failures in the testbed. A small number of failed repairs originated from patches that blocked the specific trace used by the original attacker but left alternative sequences open, which our resilient checks exposed.

Finally, compilation failures were more frequent with `DeepSeek Coder 33B` and `Qwen 3 32B`, suggesting that incomplete code fragments may be common. In certain instances, models failed silently or generated empty outputs, leading to failed test evaluations.

4.2 Example: H2O Test Case

To illustrate our evaluation process, we present the H2O case. This exploit, which occurred in March 2025, resulted in an estimated loss of about $22,700. In the H2O token, transfers originating from its liquidity pair rewarded the recipient with either H2 or O2 tokens, determined by weak on-chain randomness. Holders could then burn H2 and O2 in a 2:1 ratio to claim H2O from the token contract's balance, analogous to the chemical reaction $2H_2 + O_2 \longrightarrow 2H_2O$.

The attacker exploited this by repeatedly pushing H2O into the pair and calling `skim()`, a function in Uniswap-style pair contracts that transfers any excess tokens in the pair to a specified address without executing a swap, effectively draining tokens without performing real purchases. Each transfer triggered the flawed reward logic, enabling the attacker to grind the randomness, redeem H2 and O2 for H2O, and sell the stolen tokens for profit [8, 17].

To ensure that LLM-generated fixes for H2O were truly robust, we adapted the experiment to tolerate variations in contract state and randomness sources. The modified proof-of-concept allowed multiple attack rounds, configurable reward farming loops, and deliberate shifts in block time and height to disturb typical random number generator (RNG) patterns. In practice, the experiment evaluates multiple scenarios, including whether push–skim rewards freely mint H2 or O2 to the attacker, and none of the evaluated models successfully repaired H2O under these conditions.

Our broader objective is to create complex experiments that automatically validate repaired vulnerabilities, eliminating the need for manual verification or reliance on LLM self-validation, both of which are prone to unreliability. By building a benchmark of replayable experiments covering multiple variations of a vulnerability, we eliminate repetitive human effort whenever a model proposes a fix, concentrating labor on designing comprehensive and reusable scenarios from the outset.

5 CONCLUSION

This work presented an executable replay benchmark and a validation framework for repairing DeFi smart contract vulnerabilities with large language models. The framework starts from real incidents and replays the original exploit on patched contracts, turning evaluation into a concrete runtime check. The pipeline retrieves sources, applies targeted prompts, synthesizes and compiles patches with consistent settings, swaps bytecode to keep stable addresses, and replays the exploit with the same entry points. Our framework measures whether the attack still succeeds, whether balances and profit signals change as expected, and whether the patched contract passes basic sanity flows. This creates a faithful test that rewards real mitigation rather than superficial edits.

Results show that closed-source models are generally more consistent, while several open models remain competitive in specific cases, especially Microsoft Phi-4, which successfully repaired 2 out of 6 cases and produced compilable results for the other instances. Challenging cases such as H2O remained unsolved under strengthened replay checks, whereas simpler access control issues, such as BTNFT, were often mitigated with minimal edits.

A practical implication follows from these results. Concentrating community effort on robust, reproducible tests for each incident, with edge-case coverage around the core vulnerability, reduces manual verification and ad hoc model checks. In effect, the test suite serves as an oracle: if the replay remains profitable for the attacker, the vulnerability is unmitigated; if the replay fails at the critical step and predefined invariants hold, there is strong evidence that the vulnerability has been mitigated, enabling faster progression to subsequent cases.

LIMITATIONS The benchmark favors incidents that are publicly documented and fully replayable, which narrows coverage and underrepresents multi-contract and cross-protocol interactions. Validation semantics focus on the original attack path, so a patch may block that path while leaving alternative routes open or introducing regressions not exercised by the trace. The current release contains a limited number of test cases because each experiment required manual adjustments to make the exploit reproducible and resilient, and local open models take significant time to run on the available hardware. Model coverage and configuration are bounded by API availability and local hardware, which shapes the set of open models and the precision of quantized runs.

FUTURE WORK. Future work will combine exploit replay with complementary checks, such as property-based tests over token accounting, differential fuzzing against pre-patch behavior where safe, and selective formal verification of critical invariants such as reserves conservation. We aim to raise state realism by importing richer snapshots, perturbing mempool timing, and randomizing schedules that influence reward logic and entropy sources, following the direction established in the H2O case. On the repair side, we plan to explore agentic pipelines that interleave static analysis, symbolic hints, iterative replay feedback, and search over patch candidates, alongside training signals that reward exploiting nonviability while preserving functional behavior. As models evolve, we will track newer releases and larger open variants under a standardized setup, ensuring results remain comparable and reproducible within the same open-source framework.

REFERENCES

[1] Aider AI. 2025. Aider LLM Leaderboards: Polyglot coding benchmark. https://aider.chat/docs/leaderboards/. Accessed: 2025-08-12.

[2] ApX Machine Learning. 2025. Best LLMs for Coding — Coding-LLMs Leaderboard. https://apxml.com/leaderboards/coding-llms. Updated: 2025-07-20; Accessed: 2025-08-12.

[3] Lucas B. Germano and Julio Cesar Duarte. 2025. A Study on Vulnerability Explanation Using Large Language Models. In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence - Volume 3: ICAART*. INSTICC, SciTePress, Porto, Portugal, 1404–1411. https://doi.org/10.5220/0013379200003890

[4] Sihao Hu, Tiansheng Huang, Fatih Ilhan, Selim Furkan Tekin, and Ling Liu. 2023. Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. In *2023 5TH IEEE INTERNATIONAL CONFERENCE ON TRUST, PRIVACY AND SECURITY IN INTELLIGENT SYSTEMS AND APPLICATIONS, TPS-ISA*. IEEE, New York, 297–306. https://doi.org/10.1109/TPS-ISA58951.2023.00044

[5] Abhinav Jain, Ehan Masud, Michelle Han, Rohan Dhillon, Sumukh Rao, Arya Joshi, Salar Cheema, and Saurav Kumar. 2023. Two Timin': Repairing Smart Contracts With A Two-Layered Approach. In *2023 Second International Conference on Informatics (ICI)*. IEEE, Noida, India, 1–6. https://doi.org/10.1109/ICI60088.2023.10421047

[6] Sabrina Kaniewski, Fabian Schmidt, Markus Enzweiler, Michael Menth, and Tobias Heer. 2025. A Systematic Literature Review on Detecting Software Vulnerabilities with Large Language Models. https://doi.org/10.48550/arXiv.2507.22659 arXiv:2507.22659 [cs]

[7] Rasoul Kiani and Victor S. Sheng. 2024. Automated Repair of Smart Contract Vulnerabilities: A Systematic Literature Review. *Electronics* 13, 19 (2024), 20 pages. https://doi.org/10.3390/electronics13193942

[8] Lunaray. 2025. H2O Hack Analysis. https://lunaray.medium.com/h2o-hack-analysis-cfd167e8e99a Medium; accessed August 14, 2025.

[9] DC Marlow. 2025. *Stolen Crypto Funds Surpassing 2024 Totals Only Halfway Through 2025: Chainalysis*. The Daily Hodl. https://dailyhodl.com/2025/07/17/stolen-crypto-funds-surpassing-2024-totals-only-halfway-through-2025-chainalysis/ Accessed August 7, 2025.

[10] Emanuele Antonio Napoli and Valentina Gatteschi. 2023. Evaluating ChatGPT for Smart Contracts Vulnerability Correction. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, Torino, Italy, 1828–1833. https://doi.org/10.1109/COMPSAC57700.2023.00283

[11] Paradigm. 2021. Foundry. https://github.com/foundry-rs/foundry. Accessed: 2025-08-06.

[12] ProLLM. 2025. StackEval Leaderboard: Evaluating LLMs as coding assistants. https://www.prollm.ai/leaderboard/stack-eval. Accessed: 2025-08-12.

[13] Vince Quill. 2025. *GMX halts trading, token minting following $40M exploit*. Cointelegraph. https://cointelegraph.com/news/gmx-v1-exchange-exploited-40-million-drained Exploit of GMX V1 decentralized exchange due to reentrancy vulnerability.

[14] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3597503.3639117

[15] SunWeb3Sec. 2025. DeFiHackLabs: Reproduce DeFi hack incidents using Foundry. https://github.com/SunWeb3Sec/DeFiHackLabs. Accessed on 2025-08-06.

[16] X. Tang, Y. Du, A. Lai, Z. Zhang, and L. Shi. 2023. Deep Learning-Based Solution for Smart Contract Vulnerabilities Detection. *Scientific Reports* 13, 1 (2023), 17 pages. https://doi.org/10.1038/s41598-023-47219-0

[17] TenArmor Alert. 2025. H2O Hack Alert. https://x.com/TenArmorAlert/status/1900525198157205692 Post on X (formerly Twitter); accessed August 14, 2025.

[18] Che Wang, Jiashuo Zhang, Jianbo Gao, Libin Xia, Zhi Guan, and Zhong Chen. 2024. ContractTinker: LLM-Empowered Vulnerability Repair for Real-World Smart Contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2350–2353. https://doi.org/10.1145/3691620.3695349

[19] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddartha Venkat Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2025. LiveBench: A Challenging, Contamination-Free LLM Benchmark. In *The Thirteenth International Conference on Learning Representations*. Curran Associates, Singapore, 37 pages.

[20] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *ICLR 2025*. Curran Associates, Singapore, 55 pages. https://openreview.net/forum?id=YrycTjllL0 Oral presentation; includes benchmarking dataset and leaderboard.

[21] Lyuye Zhang, Kaixuan Li, Kairan Sun, Daoyuan Wu, Ye Liu, Haoye Tian, and Yang Liu. 2025. ACFix: Guiding LLMs with Mined Common RBAC Practices for Context-Aware Repair of Access Control Vulnerabilities in Smart Contracts. *IEEE Transactions on Software Engineering* Early Access (2025), 1–21. https://doi.org/10.1109/TSE.2025.3590108