A Methodology for Replicating Historical Exploits on EVM-Compatible Blockchains

Zhiyang Chen*, Phillip Kemper*, Yi Liu†, Jan Gorzny*, Diego Siqueira*, Yuekang Li‡,
Donato Pellegrino*, and Martin Derka*
*Zircuit
Email: {jeff, phillip, jan, diego, donato, martin}@zircuit.com
†Quantstamp
Email: yi@quantstamp.com
‡University of New South Wales
Email: yuekang.li@unsw.edu.au

***Abstract*—Replicating historical exploits on blockchain platforms is essential for testing emerging real-time defense mechanisms. However, existing tools primarily fork blockchain states and replay original exploit transactions rather than replicating these hacks in new blocks. This paper introduces a methodology for replicating exploit transactions across EVM-compatible blockchains, enabling testing of new security measures. By addressing key challenges in address mapping, contract deployment and storage configuration, our approach successfully replicates 18 security challenge exploit transactions and 12 real-world exploit transactions. This evaluation results confirm the methodology's effectiveness in recreating both controlled challenges and real-world hacks, marking a significant advancement in smart contract security research and testing.**

***Index Terms*—Blockchain, Smart Contract, Ethereum, Transaction, Fork, Testing, Ethereum Virtual Machine.**

I. Introduction

Blockchain technology has revolutionized financial transactions and data storage by introducing a decentralized, resilient, and programmable ledger that operates globally. Unlike traditional financial systems, which rely on centralized intermediaries, blockchain systems allow users to transact directly and transparently with one another. The introduction of smart contracts [1] on platforms such as Ethereum [2] has further expanded blockchain capabilities, enabling users to deploy deterministic programs that execute directly on-chain, removing the need for a trusted third party. These smart contracts serve as the backbone for a diverse range of decentralized applications (DApps) in areas such as finance, gaming, and supply chain management, forming a dynamic ecosystem of programmable digital assets. As of November 10, 2024, the Total digital asset Value Locked (TVL) across $4,210$ DApps has reached more than $\$104$ billion USD [3], underscoring the economic significance and rapid adoption of these decentralized financial systems. The rapid growth of blockchain technology has also introduced new security challenges, as malicious actors exploit vulnerabilities in smart contracts and blockchain infrastructure. Also as of November 10, 2024, financial losses from security breaches in DeFi protocols have surpassed $\$9.04$ billion USD [3], highlighting an urgent need for improved security mechanisms.

To address these security threats, various approaches have emerged to detect and mitigate attacks on smart contracts. Static analysis tools [4]–[6] are widely used to scan smart contracts for vulnerabilities, while numerous fuzzing tools [7]–[14] have been developed to stress-test contract behavior. In particular, ityfuzz [14] applies fuzzing to newly deployed smart contracts, detecting vulnerabilities as each new block is generated [15]. Meanwhile, proactive defenses such as front-running prevention techniques [16]–[18] protect users from value extraction by bots and malicious actors within the mempool. Another promising approach operates at the sequencer level; for instance, Zircuit's "Sequencer Level Security" (SLS) [19] introduces security checks directly within the sequencer layer to detect and quarantine malicious transactions before they are included in the blockchain.

However, testing the effectiveness of these novel real-time security mechanisms on existing or new blockchains remains a significant challenge. Security tools deployed on live blockchains often cannot be fully evaluated until a new attack happens, limiting the ability to measure their responsiveness under actual conditions. Although past hacks offer valuable case studies, until today they are typically used by forking the specific block where the hack occurred, which does not replicate a live environment with real-time block generation. As a result, this testing approach cannot assess a security mechanism's ability to detect an attack dynamically as it unfolds. Moreover, newer blockchains with brand-new security mechanisms often lack a history of hack transactions on their chains, creating an urgent need to replicate known exploits from established blockchains. By forking these exploit transaction, these platforms can better evaluate their defenses under realistic conditions.

To the best of our knowledge, this paper introduces the first methodology designed specifically for replicating historical exploit transactions across EVM-compatible blockchains, enabling comprehensive testing of security mechanisms in controlled environments. Our approach redeploys the contracts involved in the original exploit transaction, ensuring high fidelity to the authentic exploit. Although this paper centers on replicating historical exploit transactions, our methodology is versatile and can be adapted to replicate any other types of transaction, which are typically much simpler than exploit transactions. The remainder of this paper is organized as follows. Sections II and III cover related work on transaction replication and provide essential background information. Section IV outlines our methodology for replicating historical exploits. Section V discusses the limitations of this approach. Finally, Section VI concludes this paper.

II. Related Works

EthReplayer [20] and ContractVis [21] both provide tools for replaying historical transactions of smart contracts. However, they are limited to replaying past transactions and do not support real-time replication or cross-chain forking of transactions, features that are central to the approach proposed in our methodology. Other industrial tools, such as BlockSec Fork [22] and Tenderly Fork [23], offer users the ability to replay transactions to observe and debug each execution step. However, these tools are designed for debugging existing transactions and do not address the replication of transactions into new blocks or across chains, a capability central to the approach presented in this paper. Ressi et al. [24] review the advancements in AI-enhanced blockchain technology. Despite their insights into optimizing blockchain through AI, their discussion remains conceptual and does not detail practical real-time replication or cross-chain transaction capabilities as our work proposes.

III. Background

A. Smart Contract and Ethereum Virtual Machine (EVM)

Smart contracts are self-executing programs that automate trustless transactions directly on blockchains, removing the need for intermediaries. Foundational contracts, like stablecoins and oracles, can serve as building blocks for more complex applications. **Proxy-implementation contracts** (or **upgradable contracts**) allow smart contracts to adapt by delegating function calls through a proxy to an implementation contract, which holds the executable logic while the proxy maintains modifiable storage. This design enables redeployment and updates, which is essential for forking contracts to new blockchains. The **Ethereum Virtual Machine (EVM)** is a Turing-complete runtime environment for executing smart contracts on Ethereum and EVM-compatible blockchains, interpreting bytecode from programming languages like Solidity [25]. Our methodology can be applied to any EVM-compatible chain.

B. Types of Smart Contracts in an Exploit Transaction

Within the EVM, all program logic is encapsulated in smart contracts. Different types of contracts participate in exploit transactions, each of which requires distinct handling in the methodology described later in Section IV.

- **Hacker Contracts:** Typically closed-source, these contracts execute complex logic, such as reentrancy callbacks. Hacker contracts often include anti-front-running tactics, such as enforcing specific `tx.origin` or `block.timestamp` conditions, which prevent straightforward execution of re-deployed contracts and require bytecode modification to bypass. More sophisticated anti-front-running mechanisms may involve multiple contracts and complex logic, as detailed in prior works [26], [27].
- **Victim Contracts and Other Contracts:** Victim contracts belong to targeted protocols that lose funds during an attack and may be closed- or open-source. Other contracts, including foundational contracts, may also be closed- or open-source. They are handled similarly within our methodology.

C. Key Challenges

The simplest approach to replicate hacks would be to redeploy the same set of contracts with identical storage configurations and then execute the exploit. However, this naive approach presents three key challenges:

Challenge 1: Address Changes. Redeployed contracts cannot retain their original addresses, whether they are redeployed on the same blockchain as the original hack or on a different one. This necessitates updating all address dependencies stored in both the contract bytecode and storage.

Challenge 2: Deployment Failures. Certain contracts have constructors that depend on specific conditions, such as being deployed by particular addresses, which can lead to failures when re-deployed.

Challenge 3: Contract Storage Configuration. Changes in addresses disrupt storage configurations, particularly for slots calculated through address hashing, complicating efforts to replicate the original storage states.

In the following section, we outline how our approach addresses these challenges.

IV. Methodology

Replicating a hacker contract without source code has been thoroughly explored in previous front-running studies [26], [27]. These works focus on replicating hacker contracts, removing anti-front-running mechanisms, and executing the modified contracts to front-run the original attack. Additionally, community efforts to replicate and open-source hacker contracts in significant blockchain hacks have been notable. For instance, DeFiHackLabs [28] is a GitHub repository that has cataloged and open-sourced 539 hack contracts since November 6, 2017. In this methodology, we assume that the hacker contract has already been successfully replicated with all anti-front-running mechanisms removed.

However, even with access to the hacker contract, replicating the hack transaction poses considerable challenges. Typically, replicated hacker contracts are tested by forking the specific block at which the hack occurred. In our case, we aim to re-deploy all involved victim and other contracts at new addresses and configure their storage accordingly. This setup requires us to address Challenges 1-3 mentioned previously.

Given the original transaction hash and access to the hacker contract, we propose the following methodology to replicate the hack transaction on a new blockchain. As illustrated in Figure 1, our approach replaces the original hacker's address with a new address, termed the "Blockchain Tester." The closed-source hacker contract is substituted with an open-source "hacker replay" contract, sourced from front-running studies or open-source repositories. The victim and other infrastructure contracts are re-deployed at new addresses, with storage configurations adjusted to reflect these changes. Finally, we execute the replay contract to reproduce the hack transaction under these modified conditions.

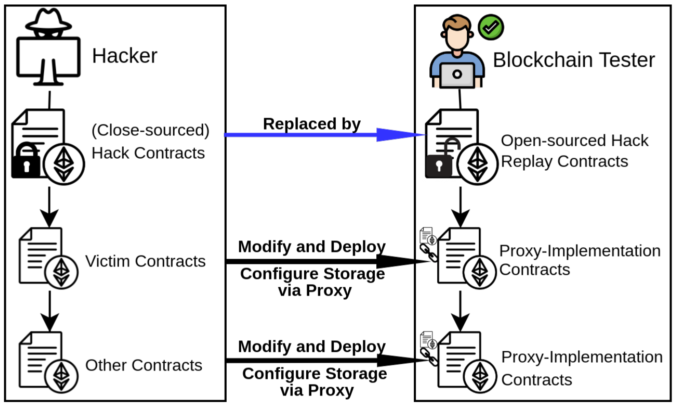
Hacker
(Close-sourced) Hack Contracts
Victim Contracts
Other Contracts
Replaced by
Modify and Deploy
Configure Storage via Proxy
Modify and Deploy
Configure Storage via Proxy
Blockchain Tester
Open-sourced Hack Replay Contracts
Proxy-Implementation Contracts
Proxy-Implementation Contracts

Fig. 1. An illustration of the forking process used to replicate historical transactions in real-time across EVM-compatible blockchains.

A. Procedure

Step 1: Track Storage Accesses with Hack Transaction. To replicate a given hack transaction, we begin by analyzing its invocation tree to identify all contracts and storage slots accessed during the hack. We track only the initial storage reads (`SLOAD`) that have not been previously read or written. Formally, each tracked storage read can be represented as ⟨`contract`, `slot`, `value`⟩. If the value is derived from an address, we document the function used to calculate it, e.g., ⟨`contract`, `slot`, `f(addressA)`⟩. For instance, in an ERC20 token contract, a token holder's balance may be stored at a slot calculated as `keccak256(k.p)`, where `k` is the mapping key and `p` is the storage slot. More complex cases, such as double mappings, may also arise. Fetching invocation tree and tracking storage accesses in this manner is well-documented and has been implemented in prior work and tools [29]–[32].

Step 2: Deploy Other/Victim Contracts. To replicate the setup of the original hack, we first distinguish the hacker contracts from other contracts involved in the transaction. Using Etherscan [33] labels like "deployed by XX Exploiter," we identify hacker contracts, while all other contracts are classified as victim or other contracts. To address Challenge 2, we replace each contract's constructor with an empty constructor to prevent deployment failures. For closed-source contracts, this is achieved by modifying the bytecode to remove the constructor. Once deployed, we link each implementation contract to a proxy contract using the proxy-implementation pattern. The proxy contract allows arbitrary storage modifications, enabling us to accurately configure states in the next step. Additionally, we use the `CREATE2` opcode to deploy contracts at predefined addresses, allowing us to know the contract addresses in advance and set the implementation address in the proxy contract accordingly. A one-to-one mapping is then established between each original contract and the proxy contract to its re-deployed contract, enabling us to replace hard-coded addresses in the original bytecode with updated addresses in the replicated setup (for instance, in Solidity, constant addresses are embedded directly in the bytecode).

Step 3: Deploy the Attacker Contract. With the victim and infrastructure contracts configured, we proceed to deploy the hack replay contract. In this step, all original addresses within the replay contract are replaced by the corresponding proxy contract addresses, as per the one-to-one mapping established in the previous step. This ensures that the replay contract interacts with the replicated setup accurately, maintaining consistency with the original exploit structure.

Step 4: Configure the Storage for Other/Victim Contracts. In Step 1, we recorded all storage reads in the format ⟨`contract`, `slot`, `value`⟩ or ⟨`contract`, `slot`, `f(addressA)`⟩. We also established a one-to-one mapping between each original contract and its proxy contract in Step 2, and identified the addresses of the open-source replay contracts replacing the original hacker contracts in Step 3. With this information, we can now configure the storage for the victim and other contracts. For each recorded storage slot, we set its value to match the value captured in Step 1. For storage slots derived from an address, we replace `addressA` with the corresponding proxy contract address and recalculate the slot value. Similarly, if `addressA` represents the hacker contract, we substitute it with the newly deployed replay contract. This process is repeated for each victim and infrastructure contract to replicate the exact storage configuration recorded in the original hack.

Step 5: Execute the Attack. In the final step, we initiate the attack by invoking the relevant functions in the hack replay contract, replicating the exact transaction sequence from the original exploit.

B. Evaluation

We implemented a prototype of the proposed approach and applied our methodology to replicate 18 hacks from the Damn Vulnerable DeFi security challenges [34] and 12 real-world hacks, as listed in Table I. We showcase[1] our methodology with the replication of a real-world hack.

V. Limitations

Our methodology faces limitations. It cannot accurately replicate hacks caused by real-time blockchain properties like `block.timestamp`, which are unique to specific blocks and blockchains. Additionally, replicating exploits involving significant amounts of blockchain-native tokens, such as ETH, could be prohibitively costly and impractical. Our approach also does not account for variations between blockchains, such

[1]Hack demonstration is available at https://github.com/jeffchen006/Forking-Exploit-Transaction-between-Blockchains.

TABLE I
REPLICATED HACKS USING OUR PROPOSED METHODOLOGY

Hack Name	Date	Blockchain
Eminence	2020-10-25	eth
VisorFi	2021-12-21	eth
SMOOFSStaking	2024-02-28	polygon
Juice	2024-03-09	eth
FIL314	2024-04-12	bsc
NGFS	2024-04-25	eth
RedKeysCoin	2024-05-27	bsc
MetaDragon	2024-05-29	bsc
JokInTheBox	2024-06-11	eth
Lifiprotocol	2024-07-16	eth
YodlRouter	2024-08-14	eth
Bedrock_DeFi	2024-09-26	eth

as network congestion, mempool dynamics, and gas prices. We focus solely on forking transactions between blockchains, thus excluding these elements from our scope.

VI. CONCLUSION AND FUTURE DIRECTIONS

This paper presents a new methodology for replicating historical exploits on EVM-compatible blockchains, addressing key challenges such as address dependencies, storage configurations, and contract deployment. Our approach facilitates accurate replication of exploit transactions, enhancing the evaluation of blockchain security mechanisms. Despite challenges like handling real-time blockchain properties and high native token costs, this methodology advances blockchain security research significantly. Future work could extend the methodology to non-EVM chains and explore generating test cases for new DeFi protocols by replicating transactions from established DeFi systems deployed on different chains.

REFERENCES

[1] N. Szabo, "Smart contracts: Building blocks for digital markets," 1996.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014, ethereum Project Yellow Paper, https://ethereum.github.io/yellowpaper/paper.pdf. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[3] "DeFiLlama," https://defillama.com/, 2024, DeFi Overview.

[4] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of Ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[5] *Securify*, Sofware Reliability Lab, 2019. [Online]. Available: https://securify.ch/

[6] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.

[7] B. Jiang, Y. Liu, and W. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269.

[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *International conference on principles of security and trust*. Springer, 2017, pp. 164–186.

[9] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for Solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[10] *Echidna*, Trail of Bits, 2019. [Online]. Available: https://github.com/trailofbits/echidna

[11] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.

[12] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[13] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan, "ModCon: A model-based testing platform for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1601–1605.

[14] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 322–333.

[15] Fuzz.land, "Fuzz.land: Platform for smart contract fuzzing," https://fuzz.land/, accessed: 2024-11-10.

[16] Z. Li, J. Li, Z. He, X. Luo, T. Wang, X. Ni, W. Yang, X. Chen, and T. Chen, "Demystifying DeFi MEV activities in flashbots bundle," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 165–179. [Online]. Available: https://doi.org/10.1145/3576915.3616590

[17] W. Zhang, L. Wei, S. Cheung, Y. Liu, S. Li, L. Liu, and M. R. Lyu, "Combatting front-running in smart contracts: Attack mining, benchmark construction and vulnerability detector evaluation," *IEEE Trans. Software Eng.*, vol. 49, no. 6, pp. 3630–3646, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3270117

[18] "MEV bot runner 'c0ffeebabe.eth' returns $5.4 million amid curve exploit," 2024. [Online]. Available: https://www.theblock.co/post/242136/mev-bot-runner-c0ffeebabe-eth-returns-5-4-million-amid-curve-exploit

[19] M. Derka, J. Gorzny, D. Siqueira, D. Pellegrino, M. Guggenmos, and Z. Chen, "Sequencer level security," *CoRR*, vol. abs/2405.01819, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2405.01819

[20] Y. Huang, R. Wang, X. Chen, C. Yang, and Z. Zheng, "Ethereum transaction replay platform based on state-wise account input data," *IEEE Transactions on Services Computing*, 2024.

[21] P. Hartel and M. van Staalduinen, "Truffle tests for free–replaying Ethereum smart contracts for transparency," *arXiv preprint arXiv:1907.09208*, 2019.

[22] "Fork - Blocksec," accessed: 2024-11-10. [Online]. Available: https://blocksec.com/fork

[23] "Forks - Tenderly documentation," accessed: 2024-11-10. [Online]. Available: https://docs.tenderly.co/forks

[24] D. Ressi, R. Romanello, C. Piazza, and S. Rossi, "AI-enhanced blockchain technology: A review of advancements and opportunities," *Journal of Network and Computer Applications*, p. 103858, 2024.

[25] "Solidity," https://docs.soliditylang.org/en/v0.8.23/. [Online]. Available: https://docs.soliditylang.org/en/v0.8.23/

[26] Z. Zhang, Z. Lin, M. Morales, X. Zhang, and K. Zhang, "Your exploit is mine: instantly synthesizing counterattack smart contract," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1757–1774.

[27] X. Deng, Z. Zhao, S. M. Beillahi, H. Du, C. Minwalla, K. Nelaturu, A. Veneris, and F. Long, "A robust front-running methodology for malicious flash-loan DeFi attacks," in *2023 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, 2023, pp. 38–47.

[28] SunWeb3Sec, "Defihacklabs," 2024. [Online]. Available: https://github.com/SunWeb3Sec/DeFiHackLabs

[29] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long, "OpenTracer: A dynamic transaction trace analyzer for smart contract invariant generation and beyond," *arXiv preprint arXiv:2407.10039*, 2024.

[30] ——, "Demystifying invariant effectiveness for securing smart contracts," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1772–1795, 2024.

[31] "Openchain," https://openchain.xyz/, 2024, accessed: 2024-11-11.

[32] "Tenderly explorer dashboard," https://dashboard.tenderly.co/explorer, 2024, accessed: 2024-11-11.

[33] "Etherscan: Ethereum blockchain explorer," https://etherscan.io, 2024, accessed: 2024-11-11.

[34] "Damn vulnerable DeFi," https://www.damnvulnerabledefi.xyz/, 2024, accessed: 2024-11-11.