LLM-Enhanced Smart Contract Fuzzing for Ethereum Pectra Security

Prayash Joshi
Virginia Tech
Blacksburg, Virginia, USA
prayash@vt.edu

Joong Hyun An
Virginia Tech
Blacksburg, Virginia, USA
joonghyun@vt.edu

Abstract

We propose a framework for smart contract security testing using Large Language Models (LLMs) for contextual mutation-based fuzzing. While traditional fuzzers lack semantic understanding of smart contracts, we want to leverage LLMs to comprehend contract context and purpose, enabling more effective test input generation. This paper presents our methodology for developing an LLM-enhanced testing framework that aims to improve vulnerability detection in Solidity smart contracts, particularly focusing on the security implications of the Pectra upgrade affecting 298.7 million addresses.

CCS Concepts

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

Keywords

Smart Contracts, Mutation, Fuzzing, Large Language Models, Ethereum, Security Testing

ACM Reference Format:
Prayash Joshi and Joong Hyun An. 2025. LLM-Enhanced Smart Contract Fuzzing for Ethereum Pectra Security. In *Proceedings of Course Project Initial Report (CS 5594).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Problem Motivation

The Ethereum network is undergoing a significant upgrade known as Pectra, affecting 298.7 million accumulated addresses. This upgrade is expected to introduce changes aimed at improving scalability, network security, and user customization. However, major upgrades such as Pectra inherently can launch with potential security vulnerabilities. To mitigate these risks, we aim to develop a test generation framework capable of identifying and addressing potential bugs as the update launches.

CS 5594, Blacksburg, VA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.2 Key Ideas

Smart contracts are self-executing programs on the blockchain that require robust security testing mechanisms. While traditional fuzzing tools like Echidna provide automated bug discovery capabilities, they primarily operate based on syntactic analysis. Our key innovation lies in enhancing mutation-based fuzzing with LLMs to incorporate semantic understanding and reasoning about smart contracts. This approach enables the development of stronger and more efficient test cases aligned with the evolving needs of smart contract security.

1.3 Proposed Contributions

This research aims to contribute the following:

- Introduce a preliminary study on the strengths and weaknesses of LLMs (both open-source and closed-source) in handling various mutations.
- Develop an LLM-enhanced mutation-based fuzzing framework for testing and validating smart contracts.
- Deploy smart contracts to Sepolia test net and check potential Pectra security issues.
- Release our LLM-based mutation testing framework for open collaborations.

2 Approach Overview

Our approach consists of two main components: First, we conduct an exploratory study to investigate the strengths and limitations of Large Language Models (LLMs) by evaluating test suites they generate against both original and mutated code. We measure coverage differences and fault detection. We want to understand how effectively LLMs handle logical variations introduced by mutations. Second, we develop a mutation-based fuzzing framework that integrates LLMs in various ways (e.g., providing one-shot examples derived from our mutation study). We then compare different LLM-based fuzzing strategies in terms of coverage and their ability to detect potential bugs.
We assess whether LLMs can generate sufficiently diverse and robust tests when presented with mutated smart contract code, focusing on coverage and the discovery of seeded faults. Our mutation-based fuzzer will leverage LLMs for test input seed generation.

3 Related Work

3.1 Traditional Testing Approaches

Fuzzing was used as a tool to detect anomalies in softwares. Engineers found that random generated input could break softwares, and the techniques were utilizied to amend vulnerabilities. Fuzzing is not limited to the traditional software development. Blockchain

developers found ways to take advantage of fuzzing, making the blockchain layers more secure and robust. [14]

3.1.1 Early Fuzzing Approaches.

Fuzzing can be divided into two eras: before AI and after AI. Before the wide usage of large language models, ContactFuzzer was developed to be one of the earliest Ethereum smart contract fuzzers. It generates inputs based on the contract's application binary interface for a valid function call and checks for vulnerabilities.[8] Echidna came two years after, an open-source fuzzer focusing on specific invariants and assertions in smart contracts.[4]

3.1.2 Fuzzing Techniques and Tools.

Fuzzing has been tailored to handle the unique sides of blockchain codes. Black-box fuzzing is generating input without the knowledge of the contract's internal execution. ContractFuzzer was developed this method. Such method can be time-consuming with invalid input attempts and may lead to a high rate of false positives unless carefully orchestrated.[8] Then there is a coverage-guided grey-box Fuzzing, which is considered a standard in smart contract fuzzing. If there is an input leading to a new path, a fuzzer further mutates it. If not, the input is deprecated. A good example is Harvey, a smart contract grey-box fuzzer which uses an algorithm to discover new input likely leading to new paths. [15]

3.1.3 Mutation Testing.

Mutation testing involves injecting small mutants into contract code, which in turn evaluates the effectiveness of the test suite. Mutation-based fuzzers start with initial set-up inputs and then produce new tests by randomly mutating the set. A study shows that mutation-based fuzzer can be more effective if sets are started from valid contract inputs.[13]

3.1.4 Generation-Based Fuzzing.

The case is different for generation-based fuzzers, which construct inputs from scratch. They refer to models or grammars of invalid inputs for generation. ContractFuzzer employs a classic generation-based fuzzing method. It utilizes type-correct but random inputs to invoke functions.[8]

3.1.5 Coverage as a Key Metric.

Coverage is often used in fuzzing to evaluate and guide smart contract fuzzers. The higher the coverage, the more likely a fuzzer is to detect anomalies in contracts. sFuzz, a grey-box fuzzer, set unexplored branches as goals to maximize coverage, while ILF(Imitation Learning-based Fuzzer) utilized coverage to prove its effectiveness to their predecessors Echidna. [11][5]

3.2 LLM Applications in Security

3.2.1 Prior to LLMs.

Before the rapid evolution of LLMs, researchers at ETH Zurich looked at combining symbolic execution with a neural network to enhance fuzzing effectiveness for Ethereum smart contracts.[6] Their framework employs symbolic execution to generate high-quality input sequences used to train neural networks to generate fuzzing policies. They were able to improve coverage and find more bugs than traditional tools, but required a large training corpus of contracts that needed to be organized into specific patterns found in training data.

3.2.2 Large Language Models as Generalized Experts.

Large Language Models (LLMs) are powerful artificial intelligence (AI) systems powered by dense neural networks that can comprehend and produce tokens at scale. Recent developments incorporate reasoning steps that allow LLMs to repeatedly call themselves, verify logic, and decide subsequent actions. Their capacity for zero-shot or few-shot learning tasks has establishes them as generalized experts in programming, writing, content evaluation, and various other tasks. Trained on extensive internet corpora containing diverse data and documentation, LLMs capture a breadth of interconnected domain knowledge that we aim to harness in our proposed framework.

In the context of Ethereum smart contracts, LLMs can generate Solidity code thanks to the rich documentation, examples, and code repositories included in its training corpus. We are interested in whether these models truly interpret the minute changes in typical syntactic Solidity code or simply produce text patterns that appear plausible. With interpretative capabilities and test input generation strategies, we can leverage the model's understanding of contract invariants, functional specifications, and potential economic exploits to better guide fuzzing analysis in high-risk code regions.

3.2.3 Recent Work.

Recent studies exploring the application of LLMs to enhance security analysis for blockchain-based systems show some good results. Chen et al.[1] provided a deep investigation of the capabilities of ChatGPT in detecting common smart contract vulnerabilities. Their findings showed that while LLMs exhibit relatively high recall for security flaws, precision scores were varied across their tests. This is likely due to misinterpretations of blockchain-specific logic. Similarly, Ji et al.[7] proposed FuzzLGen, a framework that integrates LLM agents and static program analysis to automate seed corpus generation for smart contract fuzzing. FuzzLGen significantly improves code coverage and the discovery rate of critical bugs by incorporating static program analysis into the prompting strategy. Zhang et al.[16] introduced a framework that combines LLM-based insights with rule-based reasoning to detect bugs carrying substantial monetary risk, showing how deep learning models and deterministic methods can complement each other in structured protocols.

Based on existing work, LLMs can be leveraged as "cognitive engines," interpreting code relationships and generating contextually rich input sequences to guide existing testing or scanning techniques. However, blockchain-specific smart contract development remains a niche domain, and Solidity is rarely used outside of blockchain projects. Researchers have reported measurable gains in both test coverage and bug-finding effectiveness when they incorporate LLM-based reasoning with domain-focused heuristics and examples.

3.2.4 Current Limitations.

Despite the immediate results from LLMs on certain benchmarks or repositories, there are still shortcomings such as data leakage or model hallucinations. LLM-based security analysis can also suffer from a lack of domain knowledge, particularly when dealing

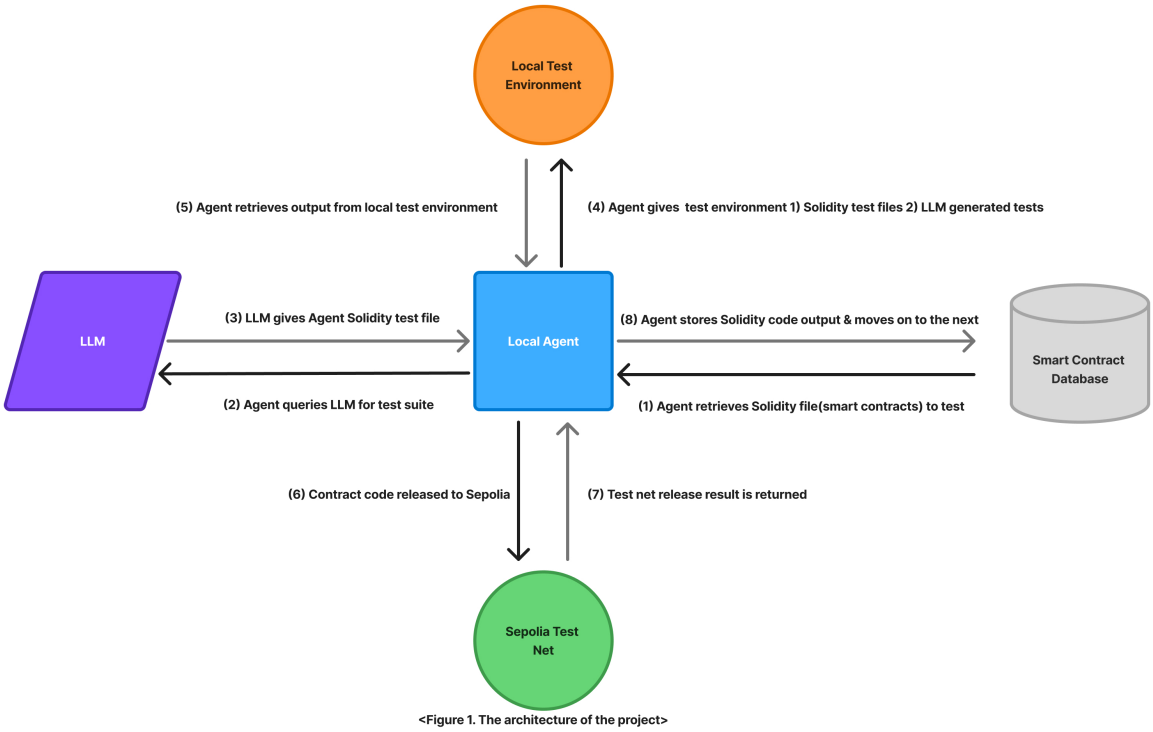

Figure 1: Project Architecture

with advanced Solidity logic or larger codebases that involve multi-layered contracts. In addition, efforts to adapt LLMs through curated datasets often hinge on obtaining high-quality labeled data.
We look to expand LLM-based test generation in smart contract domain by proposing a smart contract mutation testing framework that integrates our findings from the mutation study and strategies with targeted LLM reasoning, ensuring more transparency about its steps, while engineering the prompt to be domain-aligned.

4 Project Implementation

4.1 Architecture of the Project

Figure 1 shows the overall architecture of the project. There are five elements in the system: a local agent, smart contract database, LLM, local test environment and Sepolia test net. The local agent retrieves the Solidity file, consisting of smart contracts, from the smart contract database. The local agent then queries LLM for the test suite. Then the LLM gives the local agent Solidity file in t.sol format. The agent provides the local test environment with two types of files: a Solidity test file and LLM generated tests. When the local test environment finishes the tasks, it hands the agent with results. Then the files are deployed to Sepolia test net to check security issues. The results are returned, the output of Solidity codes are stored and the agent moves onto the next batch. The process is repeated until all data sets are tested. As both MacOS and EndeavourOS is used for development of the project, Python dependency management system is used to prevent dependency crash between two team member's development setting. Private keys and RPC URLs are stored in .env file for the duration of the final report, but will be transferred to Foundry Cast wallet for robust security.

4.2 Mutation Testing Framework

Our mutation testing framework employs a dual-pipeline architecture to evaluate the efficacy of LLM-generated test suites and to better understand how LLMs comprehend code semantics versus syntactic patterns. Figure 2 illustrates our comprehensive approach.

4.2.1 Mutation Selection.

We selected five mutation operators (aor, cor, evr, lvr, ror) following standard mutation testing practices:

- **AOR** (Arithmetic Operator Replacement): Replaces operations like +, -, *, /, and % with other arithmetic operators

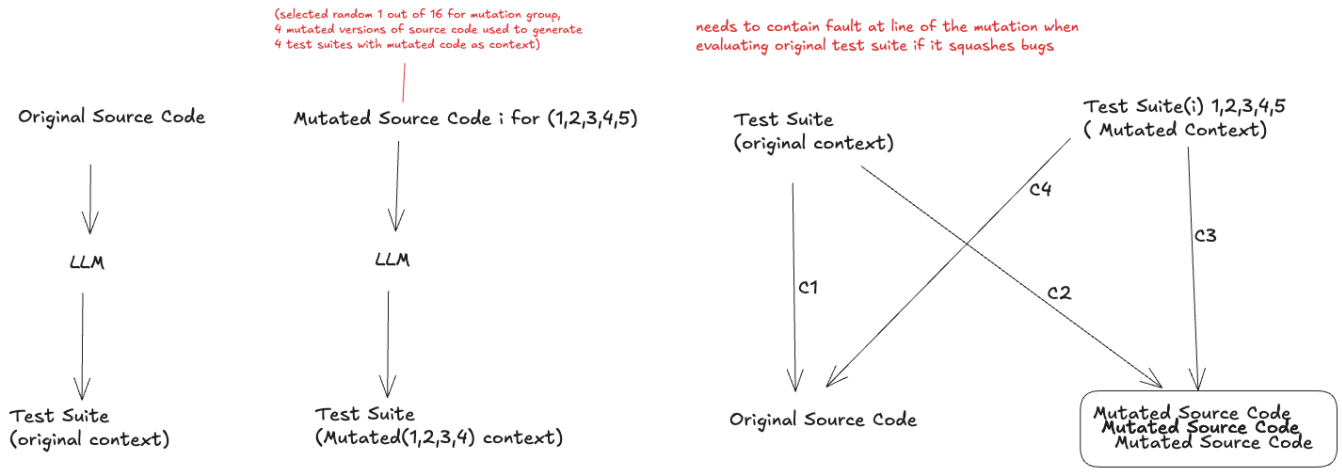

Figure 2: Two Pipelines and Evaluation Strategy

- **COR** (Conditional Operator Replacement): Substitutes conditional operators like ==, !=, <, >, <=, and >= with alternatives
- **EVR** (Expression Variable Replacement): Replaces a variable in an expression with another variable of compatible type
- **LVR** (Literal Value Replacement): Replaces literal values with other constants (e.g., changing "1" to "0" or "2")
- **ROR** (Relational Operator Replacement): Swaps relational operators in boolean expressions

For each mutation operator, we generated 16 different mutants per contract, aiming to cover different parts of the codebase. Mutations were then balanced across quartiles to ensure comprehensive coverage of the contract functionality.

We've leveraged an off-the-shelf Universal Mutator library containing vauge rules with capabilities to mutate solidity contracts. However, it lacked the fine-grained detail for distinguishing different mutation operators and provided inconsistent performance across contracts of different lengths and types. Thus, based on the mutation groups we planned aboved, we implemented our own regex rules to identify patterns and mutate. This requires a complete understanding of solidity syntax and programming practices to capture the respective pattern int the code. With our novel contribution to the existing package, we are able to provide a comprehensive look into the preformance of Large Langauge Models given differnt bugs.

4.2.2 *Two-Pipeline Evaluation Architecture.*

Our framework's novelty lies in its dual-pipeline evaluation architecture:

Pipeline 1: Original Context Test Generation.
In this pipeline, we first provide the LLM with the original, unmutated contract. The model generates a comprehensive test suite based solely on this original code. We then evaluate this test suite against:

- The original contract code (C1 scenario)
- All mutated versions of the contract (C2 scenario)

This approach tests whether LLM-generated tests are robust enough to detect subtle mutations in code, and whether they truly understand the contract's functional requirements.

Pipeline 2: Mutated Context Test Generation.
In the second pipeline, we provide the LLM with mutated contract code. Specifically, we select one random mutation from each mutation operator group (5 mutations total per contract). For each mutation, the model generates a test suite tailored to the mutated code. We then evaluate these test suites against:

- The specific mutated code used to generate them (C3 scenario)
- The original, unmutated contract code (C4 scenario)

This pipeline reveals whether LLMs adapt their test generation based on code changes, and whether they can distinguish between substantive logical changes and superficial syntactic ones.

4.2.3 *Cross-Evaluation Scenarios.*

Our framework implements four distinct evaluation scenarios to analyze different aspects of LLM test generation capabilities:

C1: Original Test Suite vs. Original Code.
The C1 scenario evaluates how well the test suite generated from the original contract performs against the original code. This serves as our baseline measurement, verifying that the LLM can produce valid, compilable tests that correctly validate the contract's expected behavior. A successful C1 evaluation demonstrates that:

(1) The LLM can generate syntactically correct Solidity test code
(2) The test suite properly understands the contract's API and expected behavior
(3) The tests successfully compile and run without errors

C2: Original Test Suite vs. Mutated Code.
In the C2 scenario, we run the test suite generated from the original contract against mutated versions of the contract. This measures the robustness of LLM-generated tests—their ability to detect semantic changes in contract behavior. A successful test should fail when

executed against a mutated contract, especially at the point where the mutation was introduced. We evaluate:

(1) Mutation Detection Rate The percentage of mutations that cause at least one test to fail
(2) Specificity: Whether test failures correspond to the specific line or function containing the mutation
(3) Coverage Differences: How code coverage metrics change when running against mutated code

C3: Mutated Test Suite vs. Same Mutated Code.
The C3 scenario examines whether an LLM can adapt its test generation when presented with mutated code. We run each test suite generated from a mutated contract against the same mutated version used for generation. This measures if the LLM:

(1) Successfully produces tests that pass when run against the mutated code
(2) Maintains code coverage despite the introduced mutations
(3) Adapts its testing strategy to accommodate the modified logic

C4: Mutated Test Suite vs. Original Code.
Finally, the C4 scenario evaluates whether tests generated from mutated code can still run successfully against the original code. This reveals whether the LLM:

(1) Focuses on testing universal contract invariants rather than implementation details
(2) Overfits tests to specific code implementations
(3) Can generalize test cases beyond the specific code version it was shown

From these four scenarios, we learn whether LLMs truly understand smart contract semantics or just pattern-match on syntax. We are aware that solidity contract syntax is very similar to JavaScript but its nuances like such as gas optimization, immutable state, and precise access control make solidity contracts a challenge in automated testing using Large Language Models.

4.3 Evaluating Tests

4.3.1 Baseline.
As a baseline, we employ Foundry (Forge) to conduct initial "sanity" tests, alongside industry-standard mutation-based fuzzing tools such as Echidna and Medusa. Widely adopted by developers, these tools provide a reliable comparison point for assessing the feasibility of context-aware, LLM-generated test suites in real-world settings.

4.3.2 Context-Fuzzer.
For our LLM-based approach, we again leverage Foundry (Forge) for basic sanity checks. We then use large language models to generate diverse and error-inducing inputs, integrating Foundry's fuzzing utilities to create a more context-aware testing system. For mutation testing, we adopt Vertigo-RS [12], a tool seamlessly integrated with Foundry, to measure how effectively the LLM-generated tests detect artificially injected faults.

4.4 Prompt Engineering/Fine-tuning LLM

Today, million dollar companies are in business based on the quality of their prompts and thier ability to leverage LLMs to their full potential to solve their user's needs. We aim to make smart-contract testing even smarter. We have employed 9 large language models comprehensive coverage-focused test suites based on smart contracts.

4.4.1 Structure.
We designed a structured prompt that guides the model through its task. In our prompt, we first assign the role of an expert Solidity test engineer to the LLM, ensuring it understands the context and objectives of generating a comprehensive Foundry test suite. We then define clear testing guidelines—including the need to cover normal operations, edge cases, and boundary conditions—and list specific technical requirements for handling Solidity constructs such as arrays and structs. Finally, we provide a standardized template format to enforce consistency in the generated code. A detailed description of this prompt structure can be found in Appendix Table **??**.

4.4.2 Advanced Context-Aware Prompting. For our final implementation, we significantly enhanced the prompt architecture with domain-specific knowledge aimed at improving test robustness and semantic understanding of solidity. In particular, we added foundry's forge techniques and "cheatcodes"[3] to the system context. We specified gas efficiency optimization instructions and emphasized minimal non-redundant and deterministic tests. Blockchain environments are difficult to test on as contracts' execution costs are directly tied to code complexity and cost of maintainence. Additionally, we included guidance for Foundry-specific testing patterns, including virtual machine manipulation commands like `vm.prank()`, `vm.expectRevert()`, and `vm.expectEmit()`. This is essential for achieving higher coverage scores for complex contracts like the ones evaluated in the final dataset. With a stricter requirements for code block delimitation through "‘`solidity` tags, we got more reliable extraction of generated code when using open-source specifically.

When comparing our basic and advanced prompting approaches revealed substantial improvements across multiple performance dimensions. The advanced prompting strategy increased successful compilation rates from 44% to 78% across all evaluated models when testing our more complex contract suite. Gas efficiency was significantly improved, with average gas consumption decreasing from 1,876,412 to 969,734 gas units per test suite execution—a 48.3% reduction. Common error patterns, particularly incorrect array access methods and undeclared variable references, were reduced. Upon analyzing the generated tests, advanced prompting produced more targeted tests that focused on critical functionality without redundancy. When evaluating SimpleNFT and BasicToken contracts, coverage remained at 100% despite using 30% fewer test functions. These results demonstrate that there are significant improvements to be made when leveraging a crafted prompts with domain-specific knowledge and structured instructions. We've enhanced LLMs' ability to generate efficient, effective Solidity test suites, particularly when evaluating potentially vulnerable mutations in complex smart contracts.

4.5 Network Test Environment

The tests are first conducted on a local host network with Foundry Forge. Once local tests are passed, the same contracts and mutated

contracts are tested on Sepolia, which is one of the biggest test nets in the Ethereum network. Sepolia is the network to have the test smart contracts distributed and nodes from Alchemy, a node-as-a-service provider, is used. Foundry Forge is used to initialize, build, create and broadcast smart contracts to Sepolia. MetaMask is used to check the balance after broadcasting contracts, and Foundry Cast is used to interact with the functions and codes released in Sepolia. Alternatively, using Remix to deploy and Sepolia Etherscan to interact with deployed codes were considered, but deprecated due to the expected time consumption of running a large number of data. Due to the limitation of gas fees, the research team may limit the number of smart contracts tested in Sepolia. The gas fees for Sepolia will be sourced from Google Cloud Sepolia Faucet, which provides users with a limited amount of gas fees every 24 hours.

4.5.1 Why Sepolia Test Net?

Various networks have been considered by the team for the main testing of this research, including the Holesky and Anvil-MetaMask networks. Sepolia was chosen because the Pectra upgrade was implemented most successfully in Sepolia, while Holesky which focused on staking update had major flaws including continuous outage, system instability, finality and client software configuration bug. Sepolia had instability issues and minor bugs at first, but is now dealt with and believed to be stabilized. Localhost network was also considered, but the idea was quickly deprecated since Localhost only reflects earlier versions of Ethereum network's structure which will not be able to fully reflect the Pectra upgrade. [9]

4.6 Dataset

Five solidity contract file in .sol format are generated through a large language model. The input prompt was to create five comprehensive and widely used contract functions that can be mutated and tested with our framework. It is important to note that these functions are for testing purposes only, and not to be used for production setting. They do simulate ERC-20 and ERC-721 behaviors, but do not fully implement their functionalities such as transferring actual ERC-20 tokens.

- **BasicToken.sol** This solidity file includes the standard ERC-20 token functionalities. The functionalities include transfer, approve, send behalf of, and track balance to handle basic transactions in ERC-20 format.
- **SimpleDAO.sol** This solidity file is a simple decentralized autonomous organization governance contract, which can be used for proposals and voting. Each proposal goes through pending, activate, succeeded/defeated, and executed. One address can vote once, with parameters such as votingPeriod that can be set for a proposal voting duration.
- **SimpleLendingPool.sol** This solidity file is DeFi lending platform for users to borrow against collateral, deposit tokens, and earn or pay interest. It allows users to liquidate undercollateralized users and also assess safety of one's position with getHealthFeactor.
- **SimpleMultiSigWallet.sol** This solidity file is creating a wallet that requires multiple signatures to execute transactions. The confirmation process is manual per user and per transaction, not an automated loop-based approval.
- **SimpleNFT.sol** This solidity file follows the ERC-721 format and allows users to use non-fungible token functions including but not limited to minting and transfer of ownership.

5 Preliminary Results

Table 1: SimpleStorage[2] Contract Test Results

Metric	Value
Project Name	SimpleStorage
Test Number	1
Gas Used	562,691
Gas Fees (in gwei)	1.000000001
ETH Amount	0.0005626910006
Block Number	1
Deployed	TRUE

Note: Gas Analysis performed in localhost anvil ecosystem

5.1 Test Compilation and Execution

To evaluate the effectiveness of the LLM-generated tests, we compiled and executed the generated test suites across different models in two phases. Our initial evaluation with simpler contracts showed promising results with certain models achieving complete success, as detailed in Appendix Table 4. When progressing to our advanced contract suite, performance varied significantly across model types and contract complexity. Our evaluation metrics included compilation success, test count, passing/failing tests, code coverage, and gas consumption. Table 6 shows the comprehensive results for our final evaluation across five complex contracts, revealing that while models like `claude-3-7-sonnet-latest` maintained high performance on moderately complex contracts, all models struggled with the most sophisticated multi-signature implementation.

We conducted a gas analysis by deploying all five contracts to the Sepolia testnet. As shown in Appendix Table 2, deployment costs varied substantially based on contract complexity, with bytecode size directly correlating to gas consumption. The SimpleMultiSigWallet and SimpleNFT contracts required the most gas (over 1.9 million units each), while the basic ERC-20 token implementation consumed just over 1 million gas units. This analysis provides valuable insight into the economic considerations of deploying LLM-generated smart contracts in production environments, particularly in the context of the Pectra upgrade where gas optimization becomes increasingly important.

5.2 Common Error Patterns

Our analysis identified distinct error patterns between the midterm and final evaluations. While initial errors (shown in Table 5) focused on basic syntactic issues like array access methods, the final evaluation revealed more sophisticated semantic challenges when testing complex contracts. Table 7 classifies these advanced errors by severity and frequency, with contract structure misunderstanding and import path errors causing the most critical failures. The SimpleMultiSigWallet contract proved particularly challenging, with all

models failing to generate compilable tests due to incomplete comprehension of its authorization flow and transaction confirmation mechanisms.

6 Project Timeline

- **2/7**: Develop initial proposal.
- **2/28**: Preliminary literature review.
- **3/6**: Initial Mutation Evaluation Tests complete.
- **3/13**: Integrate Evaluation Benchmark Dataset.
- **3/20**: Implement Preliminary LLM prompting strategies.
- **4/4**: Midterm Report.
- **4/17**: Update strategies and test on Sepolia test net for Pectra upgrade.
- **4/24**: Refine strategies based on results from the Pectra experiment.
- **5/7**: Pectra scheduled to go live on the mainnet.
- **5/10**: Final Report and Presentation.

7 Whats Next

7.0.1 *Strengths and Weakness.*

Strengths: Our preliminary findings show that Large Language Models (LLMs) can effectively generate meaningful test inputs, especially when guided by well-structured prompts.

Weaknesses: Although LLMs display promising code-generation abilities, we face recurring issues specific to Solidity and its toolchains. Some of these models were trained predominantly on more mainstream languages; thus, syntactic and semantic errors arise when handling contract-specific constructs (e.g., struct arrays, inheritance). Ensuring the generated code compiles and properly references advanced contract features is a challenge.

7.0.2 *Future Works.*

Dataset Coverage: Datasets gathered from the ASSERT-KTH/DISL Hugging Face dataset, will be used to expand our research coverage. The corresponding dataset has nearly 400,000 rows with 20 different Solidity versions with the latest one being 0.8.X. We found that more than 20,000 rows do not contain solidity version specified with Pragma. Data cleaning is handled based on a contract license, Solidity version compatibility, and contract type variety. Additional data may be collected in the future from Etherscan or Sepolia Etherscan to deal with a new type of security threat. Our work is comprehensive to cover issues arising from Pectra upgrade that may affect millions of addresses in Ethereum network. The dataset will be adding issues specific to Pectra to keep the network secure. Decomposed version is used to ensure each data set entries are unique. The smart contracts under commercial license are not published in this paper. Those that allowed the use in research will be included in the final report to showcase the effectiveness of the team's project.[10]

Cross-Evaluation Completion: Our current study provide a robust foundation through the C1 scenario evaluation. We look forward to completing the dual-pipeline architecture that looks at a comprehensive analysis of the C2, C3, and C4 scenarios. The Cross-evaluations will provide novel insights into LLMs' semantic understanding of smart contracts versus syntactic pattern-matching. The objective of the C2 evaluation is to reveal mutation detection capabilities and how effective LLM-generated tests can identify subtle semantic changes in contract behavior. Our C3 analysis will demonstrate adaptation intelligence where we evaluate if models can generate tests that accommodate mutations while maintaining coverage. And finally, the C4 scenario would demonstrate the generalization capacity of LLMs when given invariant contract mutations. We believe these evaluations will transform our preliminary findings into a comprehensive empirical study. Given our experimental infrastructure, including our contributions to the mutation generation library, evaluation metrics measurement strategy, and prompt engineering, the next step is completing the cross-evaluation matrix to advance understanding of LLM capabilities in smart contract security testing.

References

[1] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2024. When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *ACM Transactions on Software Engineering and Methodology* (Nov. 2024). doi:10.1145/3702973

[2] Cyfrin. 2023. Foundry Simple Storage Course. https://github.com/Cyfrin/foundry-simple-storage-cu. Accessed: 2025-05-08.

[3] Foundry Contributors. 2025. Cheatcodes. https://book.getfoundry.sh/forge/cheatcodes. Accessed: 2025-05-11.

[4] 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* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 557–560. doi:10.1145/3395363.3404366

[5] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 531–548. doi:10.1145/3319535.3363230

[6] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 531–548. doi:10.1145/3319535.3363230

[7] Songyan Ji, Mingyue Xu, Jin Wu, and Jian Dong. 2025. FuzzLGen: Logical Seed Generation for Smart Contract Fuzzing via LLM-based Agents and Program Analysis. Poster, Harbin Institute of Technology. https://www.ndss-symposium.org/wp-content/uploads/2025-poster-23.pdf Presented at the NDSS Symposium.

[8] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. ACM. doi:10.1145/3238147.3238177

[9] Waynes Jones. 2025. Ethereum sets new date for Pectra upgrade after technical issues. https://cryptopotato.com/ethereum-sets-net-date-for-pectra-upgrade-after-technical-issues/?amp. [Accessed: Apr, 4, 2025].

[10] Gabriele Morello, Mojtaba Eshghie, Sofia Bobadilla, and Martin Monperrus. 2024. *DISL: Fueling Research with A Large Dataset of Solidity Smart Contracts.* Technical Report 2403.16861. arXiv. http://arxiv.org/pdf/2403.16861

[11] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. arXiv:2004.08563 [cs.SE] https://arxiv.org/abs/2004.08563

[12] RareSkills. 2025. Vertigo: A Rust-based Ethereum smart contract fuzzing tool. https://github.com/RareSkills/vertigo-rs Accessed: 2025-04-04.

[13] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. LLM4Fuzz: Guided Fuzzing of Smart Contracts with Large Language Models. arXiv:2401.11108 [cs.CR] https://arxiv.org/abs/2401.11108

[14] Haoran Wu, Xingya Wang, Jiehui Xu, Weiqin Zou, Lingming Zhang, and Zhenyu Chen. 2019. Mutation Testing for Ethereum Smart Contract. arXiv:1908.03707 [cs.SE] https://arxiv.org/abs/1908.03707

[15] Valentin Wüstholz and Maria Christakis. 2019. Harvey: A Greybox Fuzzer for Smart Contracts. arXiv:1905.06944 [cs.SE] https://arxiv.org/abs/1905.06944

[16] Brian Zhang and ZHUO ZHANG. 2024. Detecting Bugs with Substantial Monetary Consequences by LLM and Rule-based Reasoning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=hB5NkiET32

A Supplemental Tables

Table 2: Gas Analysis for Contract Deployment on Sepolia Testnet

Contract	Gas Used	Gas Price (gwei)	Sepolia ETH Cost	Code Size (bytes)	Contract Address
BasicToken	1,087,466	0.001000052	0.0000010875	4,095	0x90c6...4B71
SimpleDAO	1,087,526	0.001000019	0.0000010875	4,095	0x20A2...6745
(Governance Token)	1,507,522		0.0000015075	6,316	0x8C01...72EB
(DAO Contract)	Total: 2,595,048		Total: 0.0000025951		
SimpleLendingPool	1,087,526	0.001000014	0.0000010875	4,095	0x42B2...7B32
(Collateral Token)	1,087,502		0.0000010875	4,095	0x7e07...354B
(Lending Token)	1,738,267		0.0000017383	7,285	0x3989...Dc16
(Pool Contract)	Total: 3,913,295		Total: 0.0000039133		
SimpleMultiSigWallet	1,933,784	0.001000013	0.0000019338	7,765	0x4E2D...bA76
SimpleNFT	1,945,229	0.001000013	0.0000019453	8,470	0x0095...c3aA
(Deploy)	115,913		0.0000001159		
(Mint Initial NFT)	Total: 2,061,142		Total: 0.0000020612		

Note: Gas prices shown are in gwei (10^{-9} ETH). Contract addresses are truncated for readability. Code size represents the deployed bytecode size. Total ETH cost for all contracts: 0.0000116909 ETH (approximately $0.034 at current Ethereum rates).

Table 3: Basic vs. Advanced Prompt Engineering for Solidity Test Generation

Component	Basic Prompt (Midterm)	Advanced Prompt (Final)
Role Assignment	You are an expert Solidity test engineer tasked with generating a comprehensive Foundry test suite. Focus on testing all aspects of the contract's functionality.	You are an expert Solidity test engineer tasked with generating a concise but comprehensive Foundry test suite for the specific contract. IMPORTANT: Create the MINIMUM number of tests needed for complete coverage.
Testing Guidelines	• Use the same Solidity version as the original contract • Test normal operation, edge cases, and boundary conditions • Include function-specific tests for all public and external functions • Correctly handle the return values of all functions • For arrays and structs, properly destructure return values	• Use pragma solidity from original contract • Test ONLY key functionality, edge cases, and boundary conditions • Focus on the most important public and external functions • Focus on gas efficiency and brevity • Avoid redundant and non-deterministic tests
Technical Requirements	• Use the Foundry test framework with proper imports • When accessing array items that return structs, you must destructure the result • DO NOT use dot notation on array returns	• Use Foundry test framework with proper imports • Import ONLY the contract and Test.sol • Do not create mock contracts • Use vm cheatcodes efficiently (prank, deal, warp, expectRevert) • Define only necessary test addresses in setUp() • Keep the entire test suite under 200-300 lines of code
Foundry-Specific Guidance	None provided	• Use setUp() for state initialization • Create addresses with makeAddr(): address user = makeAddr("user") • Use vm.prank() for single calls or vm.startPrank()/vm.stopPrank() for multiple calls • Test reverts with vm.expectRevert() • Test events with vm.expectEmit() • Modify timestamps with vm.warp()
Template Format	Basic template with SimpleStorage hardcoded as contract instance variable name	Enhanced template with: • Generic contractInstance naming • Pre-defined owner and user1 addresses • Event declaration placeholders • Constructor parameters placeholder • Recommendation for 8-18 targeted tests
Output Requirements	Generate only the test file code - no explanations or comments outside the code.	VERY IMPORTANT: Must enclose test suite in "'solidity and "' code blocks for reliable extraction

Table 4: Midterm Test Compilation and Execution Results by Model

Model	Compilation Success	Tests	Passing	Failing	Coverage
claude-3-5-sonnet-latest	✓	10	10	0	100%
claude-3-5-haiku-latest	✓	5	5	0	100%
qwen-2.5-coder-32b	✓	8	7	1	87%
gpt-4o-mini	✓	6	5	1	83%
gemini-1.5-flash	×	–	–	–	–
gemini-2.0-flash	×	–	–	–	–
llama-3.3-70b-versatile	×	–	–	–	–
o1-mini	×	–	–	–	–
deepseek-r1-distill-qwen-32b	×	–	–	–	–

Note: Models are sorted by test success rate and then by number of tests.

Table 5: Midterm Common Error Patterns in Failed Tests

Error Type	Description	Affected Models	Example
Array Length Access	Incorrectly trying to access `length` as a property on the array function	gemini-1.5-flash, gemini-2.0-flash, llama-3.3-70b-versatile, o1-mini	`simplestorage.listOfPeople.length`
Missing Index Parameter	Attempting to call array function without index parameter	llama-3.3-70b-versatile	`simplestorage.listOfPeople()`
Undeclared Variable	Using variable names that don't exist in the contract	deepseek-r1-distill-qwen-32b	`StoredName` instead of `storedName`
Revert Errors	Tests that compile but fail when running	gpt-4o-mini, qwen-2.5-coder-32b	`testMultiplePersons()`, `testEmptyList()`

Note: More severe errors (preventing compilation) are highlighted in darker colors.

Table 6: Final Test Compilation and Execution Results by Model and Contract

Contract	Model	Compilation Success	Tests	Passing	Failing	Coverage (%)
BasicToken	claude-3-7-sonnet-latest	✓	11	11	0	100.0
BasicToken	claude-3-5-haiku-latest	✓	7	7	0	100.0
BasicToken	o3-mini	✓	7	7	0	100.0
SimpleNFT	claude-3-7-sonnet-latest	✓	20	20	0	100.0
SimpleNFT	claude-3-5-haiku-latest	✓	9	9	0	100.0
SimpleNFT	llama-3.3-70b-versatile	✓	7	7	0	100.0
SimpleNFT	o3-mini	✓	10	10	0	100.0
SimpleNFT	gpt-4.1-mini	✓	20	20	0	100.0
SimpleDAO	claude-3-7-sonnet-latest	×	11	10	1	90.9
SimpleDAO	llama-3.3-70b-versatile	×	9	3	6	33.3
SimpleLendingPool	claude-3-7-sonnet-latest	×	16	15	1	93.7
SimpleLendingPool	gpt-4.1-mini	×	22	18	4	81.8
SimpleLendingPool	o3-mini	×	8	7	1	87.5
BasicToken	gpt-4.1-mini	×	10	9	1	90.0
BasicToken	gemini-2.5-flash-preview-04-17	×	14	13	1	92.8
SimpleMultiSigWallet	All Models	×	0	0	0	0.0
SimpleDAO	Multiple Models	×	0	0	0	0.0
SimpleLendingPool	Multiple Models	×	0	0	0	0.0

Note: Green rows indicate complete success, yellow rows indicate partial success with minor failures, and red rows indicate complete failure. The SimpleMultiSigWallet contract presented compilation challenges for all models.

Table 7: Final Evaluation: Common Error Patterns in Failed Tests

Error Type	Description	Affected Contracts	Example
Contract Structure Misunderstanding	Failure to comprehend the complete contract architecture	SimpleMultiSigWallet SimpleDAO	Error (9182): Function, variable, struct or modifier declaration expected.
Import Path Errors	Using incorrect import paths for contract files	SimpleMultiSigWallet SimpleDAO SimpleLendingPool	import {SimpleDAO} from "../../../SimpleDAO.sol";
Event Testing Errors	Incorrect implementation of event emission tests	BasicToken SimpleLendingPool	vm.expectEmit(contractInstance, Transfer(owner, user1, amount));
Tuple Unpacking Errors	Mismatch between tuple components defined and returned	SimpleDAO	(„„uint256 forVotes, uint256 againstVotes„„) = dao.proposals(proposalId);
Undeclared Identifier	Referencing events or functions that do not exist in the contract	SimpleLendingPool	emit Deposit(user1, depositAmount);
State Transition Assertions	Incorrect understanding of contract state transitions	SimpleDAO	assertion failed: Pending != Active
Function Parameter Errors	Incorrect function parameters or argument handling	SimpleMultiSigWallet	Expected ‘,’ but got ‘;’
Method Not Found	Attempting to call non-existent contract methods	SimpleLendingPool	pool.updateBorrowInterest(user1);

Note: Errors are categorized by severity and frequency. Red highlights indicate errors that completely prevented compilation, while yellow highlights indicate runtime failures. The SimpleMultiSigWallet contract exhibited the highest rate of critical errors.