Academic Year
2024-2025

Test Amplification for Smart Contracts

Jorden Van Handenhoven
Billy Vanhove

Master's thesis
Master of Science in computer science: software engineering, data science and artificial intelligence

Supervisor
prof. Prof. Serge Demeyer, AnSyMo, UAntwerpen
Supervising assistants
O. Kilincceker, M. Beyazit (AnSyMo, UAntwerpen), H. Rocha (Loyola University Maryland, USA)

University of Antwerp
Faculty of Science

Contents

List of Figures

List of Tables

Chapter 0

Preamble

English Abstract

Smart contracts, self-executing code deployed on blockchain platforms, require rigorous testing due to their immutable nature and financial impact. This thesis investigates automated test amplification techniques to enhance the effectiveness of test suites for Solidity-based smart contracts. Two complementary strategies are evaluated: search-based amplification, which uses algorithmic exploration to diversify inputs and uncover edge cases, and LLM-based amplification, which leverages large language models such as GPT and Claude to generate semantically meaningful and context-aware tests. A dataset of 117 contracts is assembled, with initial test suites generated automatically to provide a uniform testing baseline. Both amplification strategies are then applied and compared using structural code coverage as the primary evaluation metric. Results show that LLM-based methods consistently improve function and branch coverage, with Claude achieving the strongest gains. Search-based techniques, while less effective in semantic reasoning, offer valuable structural diversity and scalability. To combine their strengths, a hybrid amplification strategy is introduced, where LLM-generated tests are refined using genetic search. This yields the highest gains, achieving up to 70% branch coverage across the dataset. Along the way, this work establishes a benchmark for Solidity test amplification, provides a curated dataset, and offers insight into factors that influence amplification success. Together, these contributions support more scalable, automated, and intelligent testing workflows for smart contract development.

Nederlandstalige Abstract

Smart contracts, zelfuitvoerbare code op blockchainplatformen, vereisen grondige tests vanwege hun onveranderlijke karakter en financiële impact. Deze thesis onderzoekt geautomatiseerde test amplification technieken om de effectiviteit van test-suites voor Solidity-gebaseerde smart contracts te verbeteren. Twee complementaire strategieën worden geëvalueerd: search-based test amplification, die algoritmische verkenning gebruikt om invoer te diversifiëren en randgevallen te ontdekken, en LLM-based test amplification, die gebruikmaakt van grote taalmodellen zoals GPT en Claude om semantisch zinvolle en contextbewuste tests te genereren. Een dataset van 117 contracten is samengesteld, met automatisch gegenereerde initiële test-suites als uniforme basis. Beide test amplification technieken worden toegepast en vergeleken aan de hand van structurele code coverage als belangrijkste metriek. De resultaten tonen aan dat LLM-gebaseerde methoden consequent verbetering opleveren in function en branchcoverage, waarbij Claude de sterkste resultaten behaalt. Search-based technieken zijn minder sterk in semantisch redeneren, maar bieden waardevolle structurele diversiteit en schaalbaarheid. Om de sterktes te combineren, wordt een hybride strategie geïntroduceerd waarbij LLM-tests worden verfijnd via genetische algoritmes. Dit levert de hoogste code coverage op, met tot 70% coverage over de hele dataset. Dit werk introduceert tevens een benchmark voor Solidity-test amplification, biedt een zorgvuldig samengestelde dataset en geeft inzicht in factoren die het succes van de amplification beïnvloeden. Samen dragen deze bijdragen bij aan schaalbaardere, geautomatiseerde en intelligentere testprocessen voor smart contract-ontwikkeling.

Acknowledgments

We would like to express our sincere gratitude to our promoter, Serge Demeyer, for his invaluable guidance, expertise, and knowledge throughout the course of our thesis. His support helped us stay focused and engaged with our work.

We are also very thankful to Onur Kilincceker, Mutlu Beyazit, and Henrique Rocha, whose continuous availability and weekly meetings provided us with inspiration, clarity, and practical direction. The discussions we had with them were not only intellectually stimulating but also instrumental in helping us approach our research in an academic, structured manner.

Without their support, this thesis would not have been possible.

Chapter 1

Introduction

Blockchain technology is a distributed system that records transactions across multiple computers in such a way that the registered data is immutable and transparent. Each record, or "block," is linked to the previous one, forming a chronological "chain" that is secured through cryptographic techniques. This decentralised structure ensures that no single entity has control over the entire network, enhancing trust, transparency, and resilience against tampering. Originally devised for digital currencies like Bitcoin, blockchains are now used in a broad range of applications, from supply chain management to secure data sharing. [8]

A particularly transformative application of blockchain is the development of smart contracts. A smart contract is executable code that runs on the blockchain to facilitate, execute, and enforce the terms of an agreement between untrusted parties [9]. Deployed on blockchain platforms like Ethereum, these contracts eliminate the need for intermediaries, reduce administrative overhead, and increase the reliability of transactions. Smart contracts are widely used in decentralised applications, decentralised finance, and other domains where trust and security are critical. Most smart contracts are written in the Solidity programming language and run on immutable blockchain networks, meaning that any vulnerability or unexpected behaviour in the code can result in irreversible consequences, including financial losses or security breaches. [10, 11, 12]

Given the critical nature of smart contracts and their increasing adoption across industries, rigorous and comprehensive testing is essential. Unlike traditional applications, once deployed, smart contracts cannot be patched or modified, making pre-deployment testing the last line of defence. Unfortunately, existing testing practices often struggle to explore the full breadth of possible execution paths due to complex state dependencies, cryptographic primitives, and external interactions. [13, 14, 15]

While unit testing frameworks like Hardhat and coverage tools such as Solidity-Coverage support the development process, they still heavily rely on manual effort and the creativity of developers to design effective test cases. This often results in low coverage, poor fault detection, and missed edge cases, especially in contracts with subtle vulnerabilities. Test amplification, a subfield of automated software testing, offers a potential solution by automatically expanding existing test suites to cover more execution paths. [16, 17, 18]

Despite the promise of test amplification, its application in the domain of smart contracts remains underexplored. Two techniques, search-based and LLM-based test amplification show strong potential, yet remain underexplored in the context of smart contracts and have not been thoroughly compared. Search-based approaches, which have been studied for over a decade in traditional software systems, use optimisation algorithms to systematically explore input variations. In contrast, LLM-based methods leverage pretrained language models, such as GPT, to contextually enrich test suites by generating new test cases based on existing ones [1]. [19, 20]

To the best of current knowledge, the combination of these two approaches has received little to no attention in existing literature. The smart contract domain therefore presents a unique opportunity to investigate their effectiveness and to explore whether a hybrid method can outperform either approach individually.

This thesis investigates and compares search-based and LLM-based test amplification methods for Solidity-based smart contracts. The objective is to evaluate the impact of each approach on code coverage and testing efficiency. The search-based method utilises optimisation algorithms to mutate existing tests and generate new inputs. In contrast, the LLM-based method leverages pretrained language models to contextually generate new test cases by analysing the original smart contract and its existing test suite.

Both approaches are evaluated using a set of 117 Solidity contracts, ranging from small to higher complexity. Tools such as Hardhat and Solidity-Coverage are used to collect metrics on test effectiveness. In addition, the evaluation explores whether combining both techniques into a hybrid strategy results in improved test quality.

Based on this context, the following **research questions** are addressed:

- RQ1: How effective is search-based / LLM-based unit test amplification in increasing code coverage in Solidity-based smart contracts?
- RQ2: What are the scalability and efficiency impacts of using search-based / LLM-based amplification for smart contracts on testing time and computational resources?
- RQ3: How does search-based unit test amplification compare to LLM-based methods in terms of coverage, and efficiency for smart contract testing?
- RQ4: Can a hybrid approach combining search-based and LLM-based unit test amplification enhance fault detection and code coverage in smart contracts compared to using either method independently?

These questions guide the experimental design and evaluation presented in the subsequent sections.

This thesis makes the following key ***contributions***:

- An empirical evaluation of search-based and LLM-based test amplification techniques for Solidity smart contracts, assessing their effectiveness across multiple quantitative metrics (Chapter 5).
- The first comparative and hybrid analysis of these two approaches in the context of smart contract testing (Sections 5.7.2, 5.7.4, 5.9).
- A curated dataset of Solidity contracts and corresponding test suites to support systematic and reproducible amplification experiments (Sections 4.4, 4.5).
- A series of pilot studies to explore prompt engineering strategies and model selection for LLM-based test amplification, informing the design of the full-scale evaluation (Sections 4.6.4, 5.2.2 - 5.2.5).

The remainder of this thesis is structured as follows:
In Chapter 2, an overview of smart contracts, test amplification, and relevant tools is provided.
In Chapter 3, an analysis of related work is conducted, which can be used as reference for the thesis itself
In Chapter 4, a thorough description of the experimental setup and methods for both approaches is given.
In Chapter 5, all results from the benchmark and experiments are presented and discussed in detail.
In Chapter 6, limitations of the current setup, results and suggestions for future research are discussed.
in Chapter 7, a summary of all findings and a reflection on the implications of the results is given.

in Chapter 8, a list of all things learned through the thesis is given.
Finally, in Chapter 9, suggestions for further analysis are provided.

All chapters were jointly written by both authors. The research and implementation related to the LLM-based approach were carried out by Jorden Van Handenhoven, while the research and implementation of the search-based methods were the responsibility of Billy Vanhove. The analysis of the results, the overall writing process, and the integration of both approaches into hybrid models were collaborative efforts.

Chapter 2

Background

This thesis investigates automated test amplification techniques for smart contracts, with a focus on enhancing existing test suites through both search-based algorithms and large language models. Smart contracts, often written in Solidity and deployed on the Ethereum blockchain, present unique testing challenges due to their immutable and decentralised nature. To address these, two complementary amplification strategies are explored. Search-based test amplification modifies existing test cases using optimisation techniques to improve coverage and fault detection. LLM-based amplification, by contrast, leverages pretrained models to generate or enrich test cases based on the contract code and prior tests. These approaches represent distinct paradigms in automated testing and form the basis for the comparative analysis in this study.

2.1 Solidity Smart Contracts

Solidity is a high-level, statically typed programming language specifically designed for writing smart contracts on the Ethereum Virtual Machine (EVM). It is the most widely used language in the blockchain ecosystem due to its similarity to JavaScript and its native support for Ethereum's decentralised infrastructure. Solidity allows developers to define custom data types, manage user balances, enforce access control, and implement complex business logic all of which are critical for the correct functioning of smart contracts.

Smart contracts are full-fledged programs that are run on blockchains and enforce and execute the terms of an agreement when predefined conditions are met [10]. These contracts are deployed on a blockchain, where they operate in a decentralised and immutable environment. Solidity is well-suited for this use case because it provides native support for interacting with Ethereum-specific features such as Ether transfers, event logging, and low-level bytecode operations.

One of the key advantages of Solidity is its tight integration with the Ethereum ecosystem, including support for standards like ERC-20 and ERC-721, and compatibility with tooling such as Hardhat and Solidity-Coverage. Additionally, its support for modular and reusable code through libraries and interfaces enables the development of complex, yet maintainable, decentralised applications. Solidity's popularity, alongside its large community and open-source network, makes it so many real-world smart contracts and open-source benchmarks are readily available, which is particularly valuable for testing in academic research. [9, 21]

2.2 JavaScript for Testing Smart Contracts

JavaScript plays a central role in the testing and automation of smart contracts, particularly within development environments like Hardhat. While Solidity is used to write the smart contract logic itself, tests are typically written in JavaScript due to its versatility, familiarity among developers, and strong integration with blockchain development tools. [22]

In our thesis, JavaScript is used to define and then execute test scenarios for Solidity smart contracts. These tests interact with compiled contracts through auto generated JavaScript bindings provided by Hardhat. This setup allows developers to simulate function calls, transactions, and event emissions on a local Ethereum network that mimics the behaviour of the real blockchain.

The choice of JavaScript is also driven by its extensive ecosystem of testing libraries and assertion frameworks, such as Mocha [23] and Chai [24], which are commonly used in the Ethereum community. These libraries provide expressive syntax and tooling for writing readable tests that cover a wide range of contract interactions, from simple state changes to complex multi-user flows.

Moreover, JavaScript facilitates integration with coverage tools such as Solidity-Coverage [25], enabling developers to easily analyse how much of the contract logic is exercised by a given test suite. It also serves as the backbone for implementing automated test amplification strategies, allowing both search-based algorithms and LLM-generated test cases to be seamlessly plugged into the testing pipeline.

2.3 Hardhat and Solidity-Coverage

Hardhat is a modern Ethereum development environment designed to facilitate the building, testing, and deployment of smart contracts. It provides a very developer-friendly toolkit that supports Solidity compilation, local blockchain simulation, deployment scripting, and automated testing. Hardhat is particularly favoured in the Ethereum ecosystem due to its flexibility, speed, and tight integration with popular plugins and libraries. [26]

One of Hardhat's key features is the local Hardhat Network, which allows developers to simulate contract interactions in a controlled environment. This local network supports features like automatic state snapshots, time manipulation, and verbose error messages, making it an ideal platform for testing smart contracts. Hardhat supports writing tests in JavaScript or TypeScript, which can interact with deployed contracts through the ethers.js library. This provides a powerful interface for invoking functions and inspecting contract state. To assess the effectiveness of the test suites, Solidity-Coverage is used. It's a plugin that integrates directly with the Hardhat environment. Solidity-Coverage analyses which lines and branches of Solidity code are executed during testing and produces detailed reports with coverage percentages per contract, function, and statement. These metrics serve as a key indicator for evaluating the impact of the test amplification strategies, as they provide quantifiable insight into the extent to which the generated tests explore the contract logic. Together, Hardhat and Solidity-Coverage form the backbone of our experimental setup. Hardhat facilitates fast test execution, while Solidity-Coverage enables us to measure how much of the smart contract code is actually being tested. This combination allows us to iteratively generate, amplify, and evaluate test cases in a reproducible manner. [25, 22]

2.4 LLM-Based Test Amplification

Large Language Models (LLMs) are emerging as powerful tools for code generation and software engineering tasks. In the context of this thesis, their application is explored in test amplification, the process of enriching existing test suites by generating new test cases based on existing ones. Unlike traditional test generation techniques that rely on manual coding or heuristic algorithms, LLMs are capable of understanding the intent and structure of code, producing syntactically valid and often semantically meaningful test suggestions.

The amplification process is semi-automated: a human operator selects or writes a prompt tailored to the context of a given test file, and generates additional tests that expand the scope or explore edge cases. This generation can be done in multiple different ways, such as via an

API, a UI or a chatbot, each offering their unique advantages. These new tests are then reviewed and selectively added to the test suite. While this workflow does not achieve full automation, it leverages the LLM's reasoning capabilities to significantly reduce the creative effort required for test design. [27, 28]

This approach aligns with the objective of this thesis: to compare machine-guided test amplification strategies. Whereas search-based methods rely on algorithmic exploration of input parameters and control flow, LLM-based amplification leverages pretrained knowledge and contextual understanding to suggest diverse test cases. These generated tests have the potential to improve both code coverage and fault detection, and their effectiveness is evaluated against traditional techniques in the experimental analysis.

2.5 Search-Based Test Amplification

Search-based test amplification is an automated technique that enhances existing test suites by systematically exploring variations in inputs, parameters, or operational sequences. Its objective is to improve code coverage and fault detection capabilities by generating new test cases that exercise previously untested behaviours. These techniques typically rely on random or metaheuristic algorithms to navigate the search space of possible test modifications. Common approaches include random search, guided search, and genetic algorithms.

Random search is the most straightforward method. It generates test case variations entirely at random, constrained only by syntactic or semantic boundaries. These variations are then evaluated based on their ability to improve test metrics such as coverage or mutation score. Although random search lacks sophisticated control mechanisms, it can still be effective, especially when initialised with well-constructed baseline test cases. In this work, random search operates on existing tests by modifying inputs, control flow elements (such as function call sequences and conditions), and other structural components. All generated variants are executed in the Hardhat testing environment, with test effectiveness assessed using Solidity-Coverage. [29, 30, 31]

Guided search, while still fundamentally random, incorporates lightweight heuristics to prioritise certain mutations. Unlike genetic algorithms, it does not maintain a population or perform evolutionary operations, but it does introduce guidance on where random changes are more likely to yield beneficial outcomes. This allows guided search to explore the test space more effectively than pure random search, without incurring the computational overhead of more complex algorithms. Both random and guided approaches are easy to implement and do not require domain-specific knowledge, making them suitable as baseline techniques in comparative studies. [30, 31]

Genetic search represents a more advanced form of search-based test amplification. It maintains a population of test cases, which evolves across multiple generations. The initial generation (generation 0) consists of the original test suite. Subsequent generations are formed through two main operations: mutation and crossover. Mutation introduces random alterations to individual test cases, while crossover combines elements from multiple tests to produce new candidates. A third crucial component, selection, filters out less effective tests and retains those that contribute positively to desired metrics. Over time, this evolutionary process refines the test population, ideally leading to increased coverage and fault detection. Notably, genetic search is not limited to combining or mutating tests from the most recent generation; it may also draw from earlier generations, enhancing diversity and search effectiveness. [32]

Chapter 3

Related Work

Automated testing and test generation/amplification have been long-standing areas of interest in software engineering research, particularly in high-stakes domains where test coverage and fault detection are critical. In recent years, with the rise is use of LLMs, attention has increasingly shifted toward test amplification, the automated enhancement of existing test suites as a practical means to improve software reliability without rewriting tests from scratch. [33, 34, 20]

The following sections provide a detailed analysis about the ecosystem in which the test amplification will be performed, what tools and frameworks exist to do this test amplification and the evaluation of metrics, and commonly used approaches by other researchers to actually amplify the tests. Finally, an overview of existing datasets is given.

3.1 The Solidity Ecosystem

The Solidity ecosystem offers a mature set of tools for writing, executing, and evaluating tests. These frameworks play a crucial role in both manual testing practices and in enabling automated approaches such as test amplification.

Testing Frameworks: Several testing frameworks are commonly used in Solidity development. Hardhat provides a modern development environment that supports testing via JavaScript or TypeScript. It integrates with Mocha for structuring test cases and includes plugin support for tasks such as deployment, debugging, and tracing. Truffle [35] offers a similar test-driven workflow, also supporting JavaScript-based test writing and contract interaction. Both frameworks enable developers to simulate transactions, call contract methods, and verify expected behavior through assertions. [36]

Foundry introduces an alternative approach by allowing tests to be written directly in Solidity. This framework emphasises speed, low overhead, and developer ergonomics. It supports unit testing, fuzzing, and invariant testing, and is gaining popularity within the Ethereum developer community [37]. Remix IDE, although primarily used for interactive development and education, also includes in-browser test execution features that support rapid experimentation [38].

Coverage Measurement Tools: To evaluate how thoroughly the test suite exercises the contract code, Solidity-Coverage is widely used. It instruments contract code to track execution paths during tests and produces reports indicating which statements and branches were executed. This feedback is essential for measuring the impact of both manually and automatically generated tests. [25]

Fuzzing Tools: Fuzzing complements unit testing by generating randomised inputs that explore contract behaviour under a wide range of conditions. Echidna is a prominent fuzzing tool for Solidity. It uses property-based testing, allowing developers to define invariants that must always hold [39]. Foundry integrates native fuzzing capabilities, enabling quick specification of fuzz tests in Solidity itself [37]. Other fuzzers such as sFuzz [40] and ConFuzzius [41] combine adaptive heuristics or symbolic execution to increase the likelihood of triggering edge-case behaviours. [42]

Mutation Testing Tools: Mutation testing evaluates the effectiveness of a test suite by introducing small changes, or mutants, into the smart contract code and observing whether the tests detect the injected faults. This technique measures how well the tests can catch potential bugs by simulating realistic coding errors. In the context of Solidity, tools like SuMo [43, 44] provide automated mutant generation by applying syntactic transformations such as condition negation, operator replacement, or loop alteration. The tool then executes the existing test suite against these modified contracts. If a test fails due to the change, the mutant is considered killed; otherwise, it survives, revealing gaps in the test coverage. Mutation scores indicating the ratio of killed to total mutants serve as a robust metric for evaluating the fault-detection capability of the test suite. [45]

3.2 Evaluation Criteria for Generated Tests

Assessing the quality of automatically generated test suites whether produced by LLMs or search-based techniques requires a diverse set of evaluation criteria. These criteria range from traditional coverage metrics to more qualitative aspects such as readability and assertion quality.

Some metrics are used for both search-based and LLM-based approaches, such as code coverage, which remains a widely adopted baseline for evaluating test effectiveness across tools and studies. Mutation coverage is also frequently used to assess a test suite's ability to detect faults by measuring how well it can "kill" artificially introduced changes in the code. Additionally, test suite size serves as a proxy for conciseness and maintainability, with overly large or redundant suites often being penalised. Finally, the pass rate, the proportion of generated tests that execute successfully without errors, is a basic but important indicator of validity. [46]

Another important metric is test generation time, which reflects the computational efficiency of the amplification approach. Especially in iterative or generation-based techniques, or LLM-based approaches. The time required to produce new tests can significantly impact practical usability. Test redundancy is also a relevant factor. While related to test suite size, it specifically measures the degree to which generated tests overlap in behaviour or coverage. High redundancy may inflate the test suite without adding meaningful value, and is often seen as a sign of low diversity or ineffective generation. These metrics are increasingly recognised in both LLM-based and search-based evaluations, as they offer deeper insight into test quality beyond raw coverage numbers. [46, 47, 48, 49]

Aside from these metrics, a number of method-specific metrics are used. Some in the context of search-based techniques, some for LLM-based techniques.

LLM-Based Test Evaluation

For LLM-generated tests, syntactic validity is a fundamental requirement. Tests must compile and execute successfully to be considered viable. Most studies include a build-and-run filter in their pipeline to discard invalid test cases before further analysis. [50]

Beyond syntactic correctness, readability is increasingly emphasised as a key dimension of quality. Readable tests facilitate developer understanding and long-term maintainability. Several studies propose quantitative metrics for readability, such as metrics which assesses the semantic alignment between the test and the source code context. Empirical evaluations show that LLM-generated tests often outperform traditional tools like EvoSuite and Randoop in this regard. Additional efforts focus on refining variable names and method naming to further enhance clarity, achieving results comparable to manually written tests. [51, 52]

Another critical dimension is assertion quality. Assertions determine whether a test can effectively detect faults. While LLMs have shown promise in generating syntactically valid assertions, semantic correctness remains a challenge. Recent benchmarks, such as AssertionBench, highlight that many assertions generated by LLMs lack meaningful checks or are logically incorrect, limiting their fault-detection capabilities. [53, 54]

Prompt sensitivity is also recognised as a practical concern. Slight variations in prompt phrasing can significantly alter the resulting test code. Metrics like PromptSensiScore and the POSIX index are developed to quantify this sensitivity and guide the design of more robust prompting strategies. [55, 56]

Human usability and acceptance provide additional layers of evaluation. User studies assess dimensions such as correctness, relevance, and understandability. Findings suggest that LLM-generated tests, when well-structured, are generally accepted by developers and deemed practical for real-world use. [57]

Search-Based Test Evaluation

In the context of search-based testing, traditional code coverage metrics remain the primary benchmark for effectiveness. However, metrics like assertion quality are addressed differently in search-based frameworks. Tools like EvoSuite automatically infer assertions based on program state observations, but these assertions can be overly general or brittle. Unlike LLMs, which leverage pretrained knowledge to generate semantically meaningful assertions, search-based methods rely more heavily on instrumentation and runtime behaviour. [58]

Although readability is not traditionally a focus in search-based testing, recent research explores the integration of LLMs to post-process or rename elements in generated test suites to enhance human readability. This hybrid strategy suggests growing interest in combining both paradigms to improve test quality along multiple dimensions. [59]

3.3 LLM-Based Test Amplification

Large Language Models increasingly support the automation of unit test generation, reducing the manual effort typically associated with writing and maintaining test suites. Several tools and frameworks demonstrate the potential of LLMs to produce test code that achieves meaningful coverage. For example, TestPilot shows that LLMs can outperform traditional feedback-directed tools. In an empirical study across 25 JavaScript packages, TestPilot achieves a median statement coverage of 70.2% and branch coverage of 52.8%, compared to 51.3% and 25.6% respectively for Nessie, a feedback-directed baseline [60].

To improve LLM-based coverage further, researchers introduce methods such as HITS (High-coverage LLM-based Unit Test Generation via Method Slicing), which decomposes large methods into smaller, independently testable slices. This decomposition helps the LLM generate more focused and complete tests per slice, resulting in improved coverage for complex functions [61]. Despite these promising results, LLM-generated tests still face limitations. Studies report that a non-trivial fraction of defects remain undetected due to missing inputs, low assertion quality, or coverage gaps. One evaluation quantifies undetected bugs due to insufficient coverage between 1.81% and 6.63% across several LLMs [62].

LLMs are also explored in broader software engineering tasks beyond unit testing. Use cases include integration testing, mutation testing, and test oracle generation. Meta's Automated Compliance Hardening framework applies LLMs to guide mutation-based test generation, identifying regressions proactively [63]. Similarly, TestForge uses feedback loops between LLM-generated tests and execution results to refine test suites iteratively. TestForge achieves a pass@1 rate of 84.3%, 44.4% line coverage, and a 33.8% mutation score on the same testset on which TestPilot and Nessie were benchmarked. This shows that feedback loops between LLM-generated tests can outperform traditional tools on several metrics [50].

The suitability of LLMs for different testing tasks depends heavily on model architecture and training regime. A comprehensive benchmark across 37 LLMs shows that decoder-only models especially when fine-tuned consistently yield the best results for unit testing. Moreover, prompt engineering emerges as an effective alternative to fine-tuning, highlighting the importance of how prompts are structured during generation [64]. Lightweight program analysis techniques, as

used in tools like ASTER, further enhance LLM-guided test generation by grounding prompts in structural information such as abstract syntax trees. ASTER achieves high coverage and natural test code across multiple languages, including Java and Python. [65]

It can be observed that in many such systems, including ASTER, the focus is on aggregate metrics such as line or branch coverage, without isolating deeper factors like assertion quality or fault detection. Furthermore, since many tools use custom non-public benchmarks, reproducibility and cross-comparison remain limited. [66]

3.4 Search-Based Test Amplification

Search-based testing offers another approach for test amplification, which works by framing the test generation process as an optimisation problem. This paradigm uses metaheuristic algorithms such as genetic algorithms, simulated annealing, and hill climbing to evolve test inputs that maximise a given objective, typically code coverage or fault detection. A comprehensive survey by McMinn identifies the effectiveness of these techniques in generating high-coverage unit tests, particularly when guided by coverage metrics or fault-revealing heuristics [67].

Although initially focused on unit testing, search-based methods have been adapted to testing activities across the software lifecycle. For example, they support system-level testing, test suite minimisation, and even model-based test generation higher in the V-model. Harman et al. [68] provide evidence that search-based techniques can be scaled beyond isolated functions to support complex integration testing tasks, especially when combined with model abstractions or architectural information.

Search-based test amplification techniques often rely on either static or dynamic analysis, or a combination of both, to guide the creation of new test cases. Static analysis involves examining the source code without executing it, using syntactic and structural properties of the program. This can include analysing control-flow graphs, data dependencies, or type information. Dynamic analysis, in contrast, gathers information at runtime, such as code coverage, execution traces, or assertion results, based on concrete inputs and test executions.

Both approaches are well-represented in literature. Khamprapa, W., et al. [69] demonstrate implementations of genetic algorithms for unit testing that rely on static analysis, as well as variants that incorporate dynamic runtime feedback. Akca, S., et al. [46], on the other hand, focus exclusively on dynamic techniques to compare various test amplification algorithms, leveraging runtime metrics to guide search heuristics.

Each technique offers distinct strengths. Dynamic analysis provides fine-grained feedback from actual executions, which can improve precision when steering the test generation process. However, its effectiveness depends on the quality of inputs and test scenarios, and it is limited to the paths that are actually executed. Moreover, dynamic analysis tools often require more computational resources and complex infrastructure. This can make them harder to scale or integrate in constrained settings. [70]

Static analysis, by contrast, examines all code paths regardless of runtime conditions and can detect certain issues earlier in the development lifecycle. It avoids dependencies on test environments or runtime inputs and typically offers better scalability. However, it may lack the runtime-specific insights that dynamic methods provide, such as actual variable values or assertion outcomes. For this reason, dynamic analysis is often preferred over static analysis, as the metrics often yield higher results. [58, 69, 70]

The previously mentioned study by Akca, S., et al. even provides a direct comparison between search-based methods. It compares Blackbox Fuzzing (BF), Adaptive Fuzzing (AF), Coverage-Guided Fuzzing with SMT solver (GF) and Genetic Algorithms (GA) over a large range of metrics, such as opcode coverage, branch coverage, function call coverage, event coverage, and fault detection capabilities. Test generation time is not set as a variable, but rather as a constant in this study. They fix the test generation times to fixed numbers, and assess whether shorter or longer times have an impact on the metrics. While AF and GF consistently outperform BF and

GA over shorter test generation times (e.g. 15s), the differences in metrics minimise as these times go up (e.g. to 45s), highlighting that many of the algorithms can converge to the same solutions, when given enough time. [46]

3.5 Soldity Test Amplification

Automated test generation for Solidity smart contracts remains a relatively underexplored area compared to more mature programming ecosystems such as Java or Python. Nevertheless, a growing body of work proposes targeted approaches to improve test coverage and fault detection in Ethereum smart contracts.

One of the most prominent tools in this area is SolAR, which uses a combination of genetic algorithms and fuzzing to optimise test suites for branch coverage. Empirical evaluations show that SolAR achieves high coverage rates on real-world contracts, demonstrating the effectiveness of evolutionary strategies in this domain. [71]

SynTest-Solidity similarly applies metaheuristic search algorithms to automatically generate test cases and perform fuzzing. It integrates with the Truffle framework and offers configurable options via .syntest.js files, making it accessible for developers familiar with standard Solidity tooling. [72]

SolTG introduces a constraint-based approach by leveraging constrained horn clauses (CHCs) and SMT solvers to systematically explore the input space. This technique is particularly suited for achieving high coverage on contracts with complex logical conditions and has been shown to scale to industrial-grade smart contracts. [73]

Another notable contribution is AGSolT, which combines random and genetic algorithms for test suite generation. In addition to achieving strong coverage, AGSolT uncovers previously unknown errors in widely used open-source contracts, highlighting the practical value of automated testing in vulnerability detection. [74]

While most tools focus on traditional search-based or constraint-solving methods, PRIMG represents a newer direction by integrating LLMs into the test generation process. It uses mutant prioritisation to guide LLMs toward generating targeted and effective tests. This approach improves mutation scores while reducing test suite size, indicating that LLMs can contribute meaningfully even in highly specialised technical domains. [75]

Other efforts, such as the Solidity Test Generator (a proof-of-concept tool that relies on Mythril Classic for input generation), further illustrate ongoing experimentation in this space. However, these tools often face limitations in scalability, assertion generation, or integration with established development workflows. [76]

A more recent and scalable approach is exemplified by TestGen-LLM, developed at Meta by Alshahwan et al. [1]. This system leverages Large Language Models to propose amplified versions of existing Solidity test classes and incorporates an automated filtering mechanism that retains only those test cases which compile, pass, and increase coverage. Deployed during internal test-a-thons at Meta, TestGen-LLM demonstrated that LLM-generated test enhancements can be integrated into production pipelines and, in selected cases, surpass developer-written test suites. While promising, this study leaves open questions regarding the generalisability and coverage granularity of the improvements, motivating further research into how LLM-based approaches compare with traditional test generation techniques.

Despite these advancements, no peer-reviewed work is currently found that explores hybrid approaches combining both LLMs and search-based methods for Solidity test amplification. While prior work has explored LLM-based [1] and search-based [34, 20] test amplification independently, a systematic search using combinations of terms such as “test amplification”, “LLM”, “search-based”, and “smart contracts” in Google Scholar [77] and Semantic Scholar [78] yielded no studies applying both approaches within the smart contract domain. This lack of intersection in the literature confirms that hybrid amplification techniques for Solidity remain underexplored.

This thesis addresses this gap by evaluating both strategies individually and in combination, offering a novel hybrid framework for test amplification. This makes the work one of the first systematic investigations into combined LLM and search-based techniques for Solidity smart contract testing.

3.6 Overview of Existing Datasets

Several datasets containing Solidity smart contracts are publicly available, offering different levels of annotation and suitability for tasks such as vulnerability detection, test generation, or test amplification. While some datasets include detailed vulnerability labels, others provide test files, which are essential for evaluating test amplification methods. Additionally, certain datasets are commonly used for vulnerability research but do not include test files:

- SmartBugs Curated is developed as part of the SmartBugs framework by Ferreira et al. [79]. It contains 69 manually annotated contracts with known vulnerabilities. These contracts are selected for diversity but are not accompanied by tests.
- SmartBugs Wild, created by Durieux et al. [80], offers a much larger set of 47398 real-world contracts crawled from Etherscan. Although it provides realistic examples for static or dynamic analysis, test cases are not included.
- ScrawlD, introduced by Chavhan et al. [81], contains over 6700 smart contracts that are annotated using multiple static analysis tools. Despite its size, no test data is provided.
- Smart-Contract-Benchmark-Suites, available via GitHub by Meng, R. [82], presents a manually curated collection of annotated contracts for benchmarking purposes. However, no tests are included in this dataset either.

In contrast, some datasets provide contracts together with associated test files, making them directly suitable for evaluating automated test generation and amplification:

- The Solidity Dataset, published on Hugging Face by user Seyyed A. [83], includes 355140 contracts and tests written in Solidity, JavaScript and Python. It offers a structured format suitable for test amplification tasks.
- SolEval, created by Peng, Z., et al. [84] as a benchmark for evaluating language models pubished early 2025, comprises 1125 Solidity contracts with corresponding JavaScript tests. These examples are extracted from 9 open-source repositories, and include both the smart contracts and their original test files.

Although datasets without tests may still be useful for vulnerability detection, the presence of tests in SolEval and the Solidity Dataset makes them particularly relevant for research in test amplification. These datasets enable both input–output evaluation and the application of automated testing techniques in realistic settings.

However, based on observations during dataset exploration (see Section 4.4.4), unit test amplification tends to be more straightforward when all logic is self-contained within the Solidity contracts, that is, when they do not rely on external imports. None of the datasets above follow this structure in combination with executable tests for each contract. Most available datasets lack tests altogether. To find a suitable dataset, and extensive search via Google Scholar and Semantic Scholar using combinations of keywords such as "Solidity", "Dataset", and "Tests", was done, but without success. None of the discovered datasets featured both self-contained logic and test coverage. As a result, no public benchmark currently exists that directly supports the kind of fine-grained, framework-compatible amplification analysis targeted in this work.

Chapter 4

Method / Experimental Design

4.1 Proposed Approach

This study examines automated test amplification for Solidity smart contracts using three strategies: a search-based approach, an LLM-based approach, and a hybrid method that combines both. Each individual strategy is first assessed based on code coverage. Afterwards, the two methods are compared to understand their relative advantages and limitations. Finally, a hybrid strategy is introduced, where LLM-generated tests are refined using a search-base method, and the other way around. This step-by-step comparison helps identify the most effective and efficient amplification technique.

4.2 Evaluation Metrics

The following subsections below will provide a detailed description on which metrics are being used in the evaluation of the different methods. By default, each of the metrics is used for both search-based and LLM-based results, as well as any results obtained from hybrid models. If any of the coverage metrics is only applicable to a certain method, it will be explicitly mentioned.

4.2.1 Code Coverage

To assess the quality of the generated tests, several coverage metrics are used in addition to basic test outcomes. These include **statement**, **branch**, **function**, and **line coverage**. Such metrics show how thoroughly the smart contracts are exercised during testing. For example, high line and statement coverage indicate that most of the code is executed, while high branch coverage confirms that both true and false paths of conditionals are tested. Without sufficient coverage, defects may remain hidden in untested code paths.

The tool Solidity-Coverage is integrated into the Hardhat environment to collect these metrics. It instruments the contract code and records execution data during test runs, producing detailed Istanbul-style reports. Figure 4.1 displays a successful test execution with this setup, while Figure 4.2 shows the outcome of tests on a faulty contract. A summary of the four main coverage types for a representative contract is given in Figure 4.3. These results serve as a foundation for later analysis of the amplification techniques. [25, 36]

4.2.2 Mutation Coverage

Mutation testing is a powerful technique used to evaluate the quality of a test suite. Instead of merely measuring how much code is executed during tests as is the case with traditional metrics like line or branch coverage, mutation testing goes a step further by intentionally introducing small faults (called mutants) into the source code. These mutants simulate common programming mistakes or vulnerabilities. If the existing tests do not fail after these changes, it indicates that

```
Contract: Token
  ✓ should assign the initial supply to the deployer
  ✓ should allow valid transfer (104ms)
  ✓ should fail transfer when balance is insufficient (482ms)
  ✓ should allow transfer of exact balance (67ms)


4 passing (799ms)
```

Figure 4.1: passing tests with hardhat and solidity coverage

```
Contract: Token
  1) should assign the total supply to the wrong account
  > No events were emitted
  2) should fail on valid transfer check
  > No events were emitted
  3) should fail on exact balance transfer
  > No events were emitted


0 passing (696ms)
3 failing

1) Contract: Token
     should assign the total supply to the wrong account:

    Initial supply incorrectly assigned
    + expected - actual

    -0
    +1000

    at Context.<anonymous> (test/FaultyTokenTest.js:15:16)
    at processTicksAndRejections (node:internal/process/task_queues:96:5)
```

Figure 4.2: failing tests with hardhat and solidity coverage

the test suite is not sufficiently robust to detect certain types of errors, highlighting potential blind spots.

Unlike line coverage, which quantifies the percentage of executed lines, mutation testing provides deeper insight into the effectiveness of the test suite. Branch coverage does offer a slightly more detailed view by evaluating whether decision points (e.g., if-statements) are exercised, but it still does not assess whether the outcomes of these branches are properly verified. Mutation coverage, in contrast, assesses whether a test suite can actually detect incorrect behaviour, offering a stronger measure of test quality. Therefore, it makes a lot of sense to incorporate it as one of the metrics when assessing the test suite.

To assess the mutation coverage, the tool SuMo [44] is used. This tool is specifically designed to mutate Solidity smart contracts, and can be directly integrated within the Hardhat framework. From there, you can assess the mutation coverage for an entire test suite by running one simple command, and analysing the output. However, several practical challenges arise during this process. Although mutation testing provides informative insights, it comes with a significant computational cost. For each mutation, the smart contract must be recompiled, because the source code changed. After each new compilation, the corresponding test file must be executed. This leads to a considerable increase in total runtime. Executing the full set of mutations for a single contract can take up to an hour. Moreover, due to limitations in SuMo's current implementation, each mutation triggers the recompilation and testing of the entire contract suite, not just the contract where the mutation occurred. While compiling and testing a single contract typically takes only about 5 seconds, compiling the full suite of 117 contracts (see Section 4.4.4

```
  14 passing (5s)

-----------------|----------|----------|----------|----------|----------------|
File             |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-----------------|----------|----------|----------|----------|----------------|
 contracts/      |    85.37 |       75 |    86.36 |    89.66 |                |
  2018-10299.sol |    85.37 |       75 |    86.36 |    89.66 |... 1,16,18,297 |
-----------------|----------|----------|----------|----------|----------------|
All files        |    85.37 |       75 |    86.36 |    89.66 |                |
-----------------|----------|----------|----------|----------|----------------|
```

Figure 4.3: coverage table summary

for dataset details), followed by the execution of a full test suite, can take up to 8 minutes in some cases. 8 minutes might not seem like a long time, but this process is repeated for every mutation, making the cumulative overhead extremely high. On average, a total of around 150 mutations are introduced per contract. With a total of 117 contracts, and an average runtime of 8 minutes per mutation, the time to run the entire mutation coverage analysis in the SuMo framework for just one search-based algorithm or LLM, will take $150 \cdot 117 \cdot 6 = 140400$ minutes; 97.5 days. So while testing a small number of smart contracts, with a small number of tests might be possible, it is infeasible to do with the current setup of this thesis.

Despite its clear value as a metric for assessing test effectiveness, mutation coverage is not evaluated in this thesis due to the significant computational overhead it entails. While tools like SuMo offer valuable insights, their current implementation does not scale to the large codebases and number of mutations involved in this research, as it doesn't allow for parallel runs [43]. As a result, the importance mutation testing is acknowledged in principle, but omitted in the evaluation pipeline in favour of more practical metrics that align with the constraints of our setup. A small analysis will be conducted in Chapter 6 where the treats to validity are discussed.

4.2.3 Performance Metrics

Performance metrics are used to evaluate the impact of test amplification on the structure and effectiveness of the test suite. These metrics offer insight into how the suite evolves and how well it maintains reliability throughout amplification.

One key metric is the **test suite size**, which reflects the total number of test cases after amplification. A larger suite may indicate thorough exploration, but can also imply increased maintenance overhead. The **pass/fail rate** complements this by indicating the proportion of tests that pass successfully. A high pass rate suggests that the amplification process maintains semantic correctness, while a lower rate may highlight the presence of invalid or redundant tests.

Another important measure is the **test inflation factor**, defined as the ratio of the number of newly generated tests to the size of the original test suite. This metric captures the degree of expansion resulting from the amplification process and helps assess its scalability. All performance metrics are directly derived or can be directly computed from the Hardhat test output, where both the number of tests and their pass/fail outcomes are reported after each run.

To capture the practical trade-off between test quality and test volume, **relative efficiency** is introduced as a composite interpretative metric. It reflects how effectively a model translates amplified test output into meaningful coverage, while accounting for the pass rate and the size of the generated test suite. Concretely, efficiency is assessed by jointly considering three observed variables:

- the coverage gain achieved by the model
- the pass rate
- the test inflation factor

Although no explicit formula is used, the comparison of these variables reveals meaningful differences in model behaviour. For instance, a model achieving high coverage gains with a low inflation factor and high pass rate is considered more efficient. Conversely, high test volume with limited additional coverage and a low pass rate signals lower efficiency. This approach allows for qualitative, grounded assessment of model performance in practical, scalable settings.

LLM-based methods operate in a fairly uniform environment: same prompt structure, deterministic or probabilistic output, and limited configuration options. This makes a composite efficiency metric like "coverage gain vs. test inflation vs. pass rate" both meaningful and fair across models.

In contrast, search-based methods are fundamentally tunable systems. Parameters like: mutation rate (refers to how often input mutations occur during the search in this context), crossover strategy, population size, number of generations etc. all directly impact the number of tests generated, coverage progression, and even redundancy levels. This means that comparing efficiency across search-based approaches using the same relative metric could be misleading, unless normalisation for search effort is done. This adds unnecessary complexity to the study as the other metrics are sufficient to assess the methods [85]. However, the test inflation factor will be computed, in order to compare search-based methods to LLM-based methods.

Relative efficiency is assessed only for LLM-based methods, as these operate under comparable conditions with fixed prompt inputs and limited control over generation parameters. In contrast, search-based approaches vary significantly in configuration and generation strategy, making direct comparisons based on test volume less meaningful without normalisation for search effort.

These performance indicators, together with the coverage results, provide a comprehensive understanding of how each amplification strategy affects both the breadth and depth of the test suite.

4.2.4 Metrics not covered in the study

While several quantitative metrics are employed to assess test amplification, some relevant metrics are deliberately excluded from this study due to their subjective nature or practical limitations.

One such metric is **code readability**. This is particularly relevant in the context of LLM-generated tests, although it can also apply to search-based approaches. Readable test cases, characterised by clear structure, meaningful comments, and logical flow, are more easily maintained, extended, and understood by developers. Since amplified tests are ultimately intended for human use, high readability can significantly improve their long-term utility. However, this metric is inherently subjective. Other research often uses various ways to assess the readability, such as validation through surveys [86] and experiments to validate identified readability factors (e.g., method naming, complexity, comments ...) [87]. Moreover, some tests may lack readability yet still contribute significantly to coverage or fault detection. In this study, where the primary focus is on test effectiveness, readability is therefore considered less critical and has not been formally evaluated.

Another omitted metric is **test generation time**. While relevant for both search-based and LLM-based methods, this metric is especially pertinent to LLMs due to their variable response times. In contrast, search-based algorithms typically generate tests rapidly and deterministically. Measuring generation time could offer insights into the trade-off between efficiency and quality. However, in this thesis, test generation with LLMs is performed through manual prompting rather than automated pipelines or APIs (as discussed in Section 4.6.3). As a result, generation times are inconsistent and difficult to measure reliably. Given these limitations, test generation time was not included as a metric in this study.

One final metric that was considered but ultimately excluded from this study is **test redundancy**. Both LLM-based and search-based amplification techniques can produce redundant

tests cases that do not contribute meaningfully to coverage or fault detection. This issue is more pronounced in search-based methods, which can generate large volumes of tests rapidly, though LLMs may also produce semantically similar cases. Redundant tests enlarge the test suite, increasing complexity and maintenance overhead without providing additional value. Measuring redundancy could offer insights into the degree of test suite saturation introduced by each amplification method. However, accurately detecting redundancy is non-trivial. Automated techniques to find similar code segments can identify structurally similar tests, while executing tests and comparing their runtime paths and effects, can flag those with identical behaviour [47, 48]. Unfortunately, limitations in the Solidity-Coverage tool used in the Hardhat environment make the latter approach infeasible in this study (see Section 4.6.7). Manual review is also possible, but prone to human error and inconsistency. Due to these practical challenges, test redundancy was not adopted as an evaluation metric.

These exclusions are acknowledged to ensure transparency in metric selection and to clarify the scope within which test effectiveness is evaluated.

4.3 Setup

Building on the instrumentation framework described above, the following setup ensures a consistent environment for evaluating each amplification technique. The testbed is designed to support both coverage measurement and mutation analysis, enabling systematic comparison across search-based, LLM-based, and hybrid strategies.

The test amplification framework is organised into a modular directory structure, designed to support consistent evaluation of Solidity smart contracts across multiple amplification strategies. The environment is configured using Node.js v18, Hardhat v2.22.17, and the Solidity compiler version 0.8.24. Code coverage is measured using Solidity-Coverage v0.8.x, while mutation testing is performed using the latest available version of SuMo v2.5.4 from its official GitHub repository. SuMo is included as part of a small-scale study to demonstrate the applicability of mutation testing in this context, rather than for large-scale evaluation.

All smart contract files are placed in the contracts/ directory. Each contract is accompanied by a corresponding JavaScript test file, stored in the test/ directory. These test files are structured using the asynchronous testing utilities provided by Hardhat. The hardhat.config.js file specifies the Solidity version, testing settings, and plugin configurations required for compilation, deployment, and analysis.

When the test suite is executed using Hardhat, Solidity-Coverage generates a detailed coverage report, based on the tested contracts. This report includes statement, line, function, and branch coverage, and is saved both in the coverage.json file and visually rendered in the coverage/ folder. These metrics provide essential insights into the extent to which the test suite exercises the contract logic.

To facilitate further analysis, a custom-built utility parses the contents of coverage.json and exports the relevant data into an Excel-compatible format. This allows for systematic comparison between the original and amplified test suites based on percentage increases in coverage metrics.

The overall setup is designed for repeatable evaluation. Regardless of the amplification technique applied whether search-based, LLM-based, or hybrid the structure of the framework remains unchanged. Amplified test files are inserted into the same test/ directory, replacing or extending the original files, and are evaluated using the same unified pipeline. This modular design ensures that each test iteration can be executed, measured, and compared consistently across all experiments. [43, 22]

Figure 4.4 illustrates the complete test amplification workflow used in this study. It captures the key stages of the pipeline from taking the original smart contract and its test suite, through amplification using either an LLM- or search-based method, to the filtering of candidate tests based on build and execution outcomes. Only those test cases that can be deployed and pass

are added to the improved test suite, while failing cases are retained for further inspection as potential indicators of vulnerabilities.

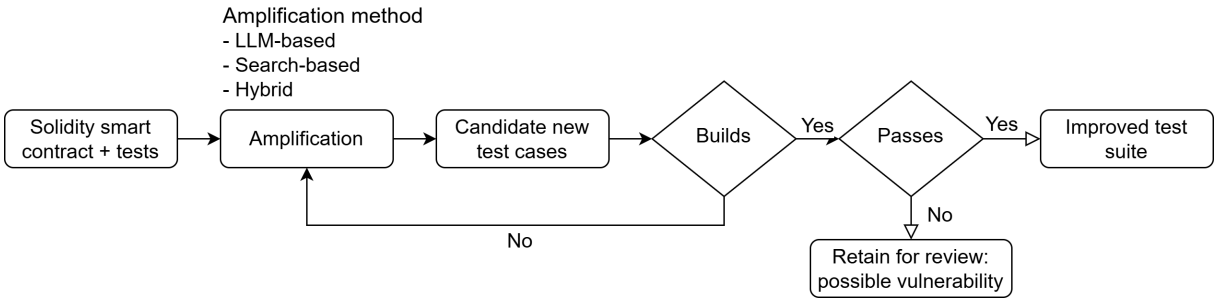


Figure 4.4: Test amplification workflow

Replication Package

To facilitate reproducibility and allow other researchers to build upon this work, a complete replication package is made available on GitHub[1], which can be found in Appendix A.1. This package includes all scripts, source code, configuration files, and generated test cases used during the experiments. A detailed README file is provided to guide users through the setup process, including the installation of dependencies, execution instructions, and explanation of the folder structure. The repository covers the following components: a full implementation of all test amplification techniques and additional logic, parameter settings used in the experiments (e.g., mutation rates, crossover thresholds, stopping criteria), all generated test cases per contract and per amplification strategy, making it possible to directly inspect output quality. evaluation scripts used to calculate metrics such as coverage, generation time, and test inflation.

Descriptions of the experimental setup, including all measurement criteria (e.g., how test generation time was measured, how failing cases were filtered, and how manual contract analysis was done), are provided in the following sections. By combining the open-source repository with clearly defined experimental procedures, this study aims to ensure transparency and full replicability.

4.4 Testbench Creation

4.4.1 Data collection from github

To initiate the process of test amplification, the first step involves identifying a suitable testbench. While no specific number of required tests was defined in advance, the objective is to obtain as many tests as possible. This is achieved by scraping GitHub repositories. GitHub allows users to search by programming language and sort results by popularity, using the number of stars. This functionality makes it easier to locate relevant tests by targeting repositories containing a significant number of Solidity files, prioritised by star count. Highly starred repositories are often widely used and tend to be large, making them suitable sources for extracting contracts and tests.

This scraping process marks the starting point of the research. Contracts and tests are collected from the three largest GitHub repositories that use Solidity files. Those are Uniswap, Synthetix, OpenZeppelin [88, 89, 90]. The plan is to gradually expand the testbench by proceeding to the fourth largest repository, then the fifth, and so on. The main goal in the beginning, is to get familiar with the tools and the Solidity language itself. Therefore, an extensive testbench is not required at the outset, which justifies the decision to limit the initial selection to the three

[1] https://github.com/jordvhan/Solidity_smart_contracts_amplifier_SB_vs_LLM/

largest GitHub repositories, and continue with the scraping process later on, when more tests are required.

4.4.2 Problems with this approach

Despite the initial success in collecting a significant number of contracts and tests, several issues emerge that limit the long-term usability of the selected testbench. A major challenge is the inconsistency in Solidity versions used across different repositories, leading to compatibility problems during compilation. Additionally, limited prior experience with the required toolchains contributes to the difficulty, as resolving these issues proves to be time-consuming and impractical within the scope of the project. After all, the focus isn't on achieving the optimal testbench, but rather on a comparative analysis between different test amplification methods.

In addition, many contracts rely on external libraries that are either outdated or unavailable, causing broken imports and unresolved dependencies. Since Hardhat, the toolchain that is used to evaluate the tests, requires all the contracts to be compiled at once, a single broken import can hinder the entire test process. These two technical obstacles significantly hinder the execution of tests. As a result, after the completion of the first iteration of the research, it is concluded that this testbench is no longer a viable basis for continued experimentation. Nonetheless, due to its role in the initial setup, it remains relevant to document this phase of the work.

Some attempts are made to manually adjust imports or downgrade Solidity versions to achieve compatibility. However, these fixes are not scalable and introduce new inconsistencies, which further reinforce the decision to discontinue the use of this testbench after the first iteration.

4.4.3 Transition to a stable testbench

Following the limitations encountered with the initially constructed testbench, an alternative testbench is sought. The goal is to find one that has previously been used in academic research and provides a more stable foundation for test amplification. During this search, a publicly available dataset containing thousands of Solidity contracts is identified [82]. This dataset is created by Meng Ren, a software engineer from China[91]. This dataset is easy to use and contains contracts from many different sources, grouped by size, date and use case, which further validates its reliability.

A key advantage of this dataset is the absence of import statements in the contracts. Moreover, any functionalities that would typically rely on external libraries are directly included within the contracts themselves, ensuring that all required logic is self-contained. This immediately resolves the issue of broken or unavailable dependencies. However, it must be noted that this setup does not reflect real-world development practices, where imports are commonly used to avoid code duplication and facilitate modular design. The implications of this limitation are discussed in more detail in Chapter 6 about treats to validity. Furthermore, Hardhat supports the use of multiple Solidity compilers simultaneously. As a result, the discrepancy in compiler versions across contracts no longer poses a problem. The lack of interdependent imports eliminates the risk of version conflicts between libraries and contracts, thereby overcoming the two main technical obstacles previously encountered. This results in a much more robust dataset, which offers a more scalable foundation, making it suitable for the subsequent stages of the research.

However, the dataset also presents a significant drawback. While the contracts themselves are highly suitable for the research goals, the dataset does not include any corresponding test files. This is likely because the dataset is primarily used in the context of test generation rather than test amplification. As a result, the absence of existing tests poses a challenge, given that test amplification methods rely on a base set of tests to expand from.

4.4.4 Generating a Custom Testbench

Initially, the idea is to manually write test cases for a selected set of contracts. However, due to the size of the dataset and the time constraints of the project, this approach quickly proves to be impractical. As a compromise between quality and scalability, it is decided to generate tests using an LLM-assisted method. Rather than aiming to generate tests for the entire dataset, a time-bound strategy is adopted: dedicating a fixed number of hours to generating tests and evaluating how many contracts can be covered within that time frame.

To implement this solution, a custom testbench is created through automated test generation. A subset of contracts is chosen, for which test cases are generated using GitHub Copilot, powered by ChatGPT-4o. This process is carried out within Visual Studio Code. Each contract is submitted individually to the tool, with a prompt instructing it to generate tests aimed at achieving full coverage. The tests are written using the Hardhat testing framework and saved in appropriately named files corresponding to each contract. Specific technical constraints are also included in the prompt, such as avoiding the use of .deployed() for contracts written in Solidity 0.4.x or 0.5.x, and always using ethers.parseEther for value handling. These measures ensure compatibility and correctness across Solidity versions. Prompt used for test generation:

Generate test cases for this smart contract try to get 100 percent coverage and write the tests in hardhat_testing/test number of the contract and then -test.js. warning: Do not call .deployed() after deploy() if the contract is written in Solidity version 0.4.x or 0.5.x. Contracts compiled with these older versions already return a deployed instance, so .deployed() will cause a TypeError. Only use .deployed() for contracts compiled with Solidity 0.6.x or higher.Also always use ethers.parseEther. Only make a test file.

The LLM-based test generation process resulted in over one thousand test cases, covering a total of 117 contract files. For each contract file, one test file is generated. However, each contract file doesn't just contain the one contract under test, but also additional helper functions and other contracts from which the contract under test is derived using inheritance. In total, the dataset contains 967 unique contract entities: 117 main contracts and 850 additional base contracts and library contracts with helper functions.

These additional components are relevant when evaluating coverage, as tools such as Hardhat include all declared contract code in their coverage analysis. Importantly, unlike traditional Solidity projects where libraries are imported in full, the dataset includes only those functions from a library or base contract that are actually used. For example, if a contract applies a SafeMath function (commonly used in Solidity smart contracts), only that specific function appears in the dataset version of SafeMath, making full coverage more attainable. This structure ensures that achieving 100% coverage is possible, as all executable logic is embedded in the contract file and directly invoked by the contract under test.

4.4.5 Evaluation of the Generated Tests

Despite the high number of generated tests, 1073 in total, averaging just over 9 tests per contract, the quality of these tests is inconsistent. An initial test run reveals that only 568 tests execute successfully, while 505 fail, resulting in a success rate of approximately 52.93%. Moreover, the failure distribution is not random: contracts typically exhibit either mostly successful or mostly failing tests. In some cases, all tests pass without issue; in others, all tests consistently fail. This pattern suggests that the LLM is able to accurately interpret and test certain contracts, while struggling with others. The variability in success may be due to differences in contract complexity, naming conventions, or how explicitly functions are defined in the code. Contracts with clear structure likely align better with the model's training data, resulting in more effective test generation. Additionally, contracts with a lot of comments also experience a higher

test success rate. Conversely, less conventional, larger, or more abstract contracts may lead to misinterpretation and thus a higher rate of invalid tests.

This uneven distribution highlights a limitation in the reliability of the LLM-generated output. Given the essential role of the testbench in the upcoming amplification analysis, it is necessary to address these shortcomings. To do so, a manual inspection is initiated for each failing test. The decision is made to prioritise efficiency: any test that can't be quickly understood and corrected is discarded rather than consuming excessive time. This approach allows for the retention of as many valid tests as possible, while the LLM-generated output requires significant human intervention to reach a usable level of quality.

4.4.6 Finalising the Testbench for Amplification

Out of the 117 contract files initially targeted, 113 can be successfully tested. Four contracts have to be excluded entirely due to specific technical limitations. The issues encountered are as follows: One contract file exceeds 24kB in size, which makes it impossible to deploy within the test environment, making it untestable. Another contract defines a complex game system with worlds and characters, requiring multiple deployments and interdependent logic. The file's size and structural complexity make it incomprehensible to the LLM and equally infeasible for manual resolution. The third contract contains a contract that's over 1100 lines long. The constructor constructs several different objects and initialises dependencies between them. This constructor takes various parameters but neither the LLM, or manual effort, is able to successfully construct the contract. The final excluded contract has ten generated tests, all of which fail for unclear reasons. While the contract can be successfully deployed in the test environment, neither manual inspection nor further assistance from the LLM yields any workable fixes. Despite these four omissions, the refined testbench remains a substantial and diverse foundation for the amplification experiments to follow.

In summary, following the manual correction process, a total of 923 tests are retained from the original 1073, meaning that 150 tests are discarded. These rejected tests are either too time-consuming to fix or prove unresolvable even after several attempts. Some tests experience additional problems, making them unfixable. As a result, the final testbench consists of 923 validated tests.

Descriptive statistics of the testbench

To assess whether the generated tests form a meaningful and representative benchmark for evaluating test amplification techniques, several descriptive statistics of the testbench must be examined. Beginning with the distribution of achieved coverage values prior to amplification.

In the box plot in Figure 4.5, it becomes apparent that the achieved coverage by the original test suite varies substantially. As expected, this variation reflects differences in contract complexity and in how thoroughly the original tests were generated. The box plot of the initial statement coverage values suggest a roughly even or mildly normal distribution, with no excessive skew or clustering. This supports the claim that the testbench does not overrepresent either trivial or overly complex contracts, but instead provides a balanced mix that mirrors realistic testing scenarios. As for the other coverage metrics, the results seem to be well distributed. While it doesn't exactly follow a normal distribution, it comes relatively close, which shows that the testbench can be considered representative for downstream test amplification. While no benchmark can be truly universal, the current one captures sufficient heterogeneity to generalise conclusions about amplification performance across diverse Solidity contracts.

To further confirm this statement, two more plots are shown. Figure 4.6 shows the statement and branch coverage plotted to the lines of code. For clarity, only the scatter plots for branch and statement coverage are shown in the main text, as these are the most indicative of structural test quality. The remaining coverage metrics exhibit similar patterns and are included in Appendix A.2 for completeness. In each of these scatter plots, the data points appear widely

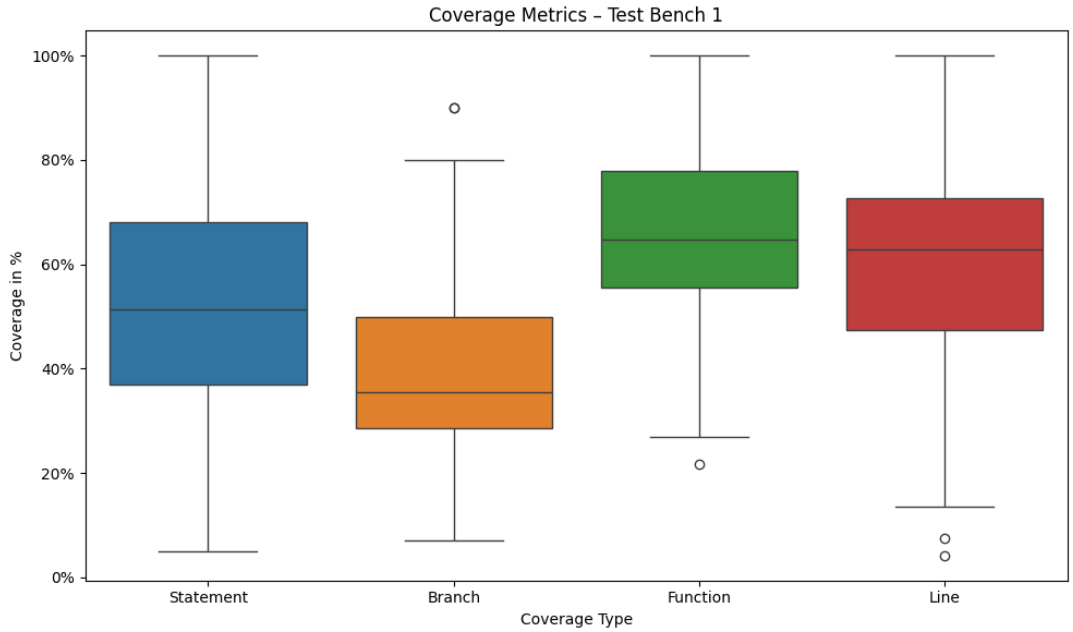


Figure 4.5: Coverage distribution testset 1

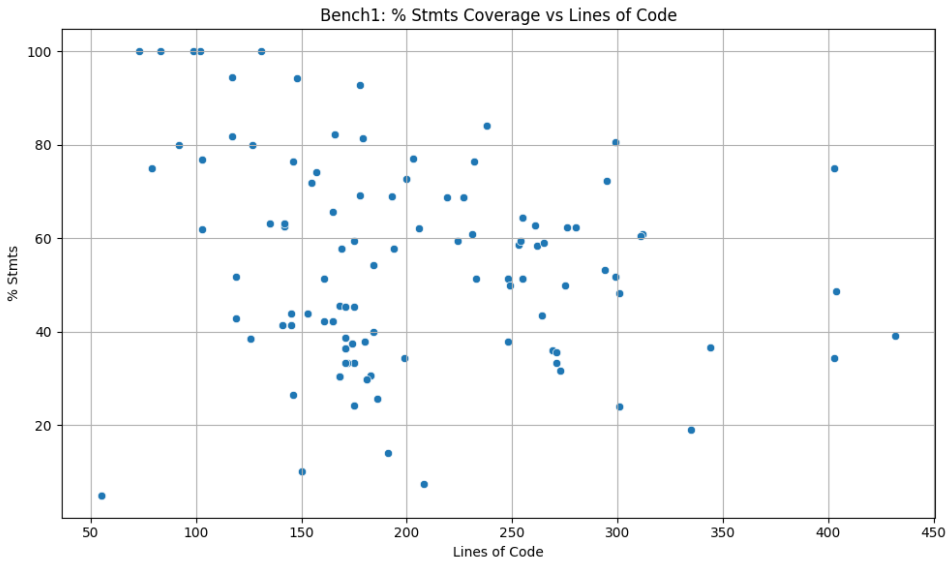
dispersed without any visible trend or correlation. This observation is important because it indicates that the achieved coverage is not simply a function of code size. In other words, larger test files do not systematically achieve higher or lower coverage than smaller ones. This reinforces the claim that the testbench provides a structurally diverse and balanced dataset, where coverage results are influenced by meaningful factors such as control flow and test design, rather than being biased by test file length.

4.4.7 Issues Encountered During Test Evaluation

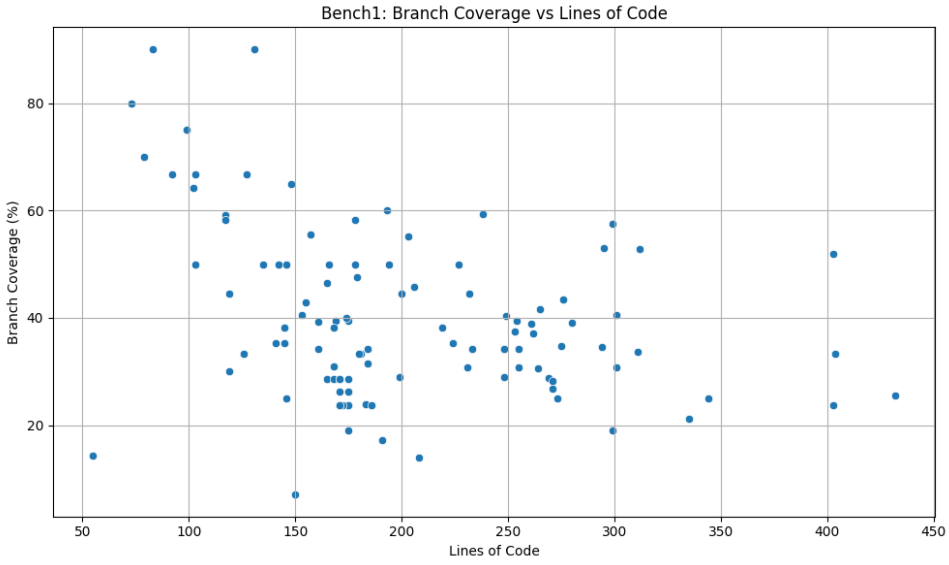
During the cleanup phase of the testbench, a variety of recurring issues are identified in the generated tests. These problems are categorised based on whether they could be resolved with minimal manual intervention, or whether they prove too complex within the scope of the project, or take too much time to efficiently resolve.

In case of the simple issues: Several problems are straightforward to detect and correct, and therefore do not hinder the inclusion of the corresponding tests:

- Outdated syntax: A frequent issue is the use of ethers.utils.parseEther() instead of the newer ethers.parseEther(). The former syntax is deprecated in newer versions of Hardhat and Ethers.js, but the fix is trivial and is applied consistently.
- Incorrect argument types: The LLM occasionally passes arguments of the wrong type. For instance, some functions expect a list (e.g. [100]) rather than a scalar value (e.g. 100). In such cases, the fix simply involves correcting the argument structure.
- Incorrect decimal assumptions: Many smart contracts define a decimals property, often set to 18 to mimic Ethereum's smallest unit (wei), where 1 ETH = 10^{18} wei. However, some contracts in the dataset define a different number of decimals (e.g. 10). The LLM often assumes 18 by default, which causes mismatches in test expectations. These issues are resolved by aligning the tests with the correct decimals setting from the contract.
- Testing of private logic through public wrappers: Some tests attempt to directly call private functions, which is not permitted. When the contract provides a public function that internally calls the private one (e.g. performing a small extra operation before invoking the internal logic), the test is rewritten to use the public wrapper instead.



(a) Statement coverage to LOC



(b) Branch coverage to LOC

Figure 4.6: All coverage metrics plotted to the Lines Of Code (LOC)

In case of the complex issues: Certain problems can't be resolved within the constraints of the research project, either due to Solidity limitations or inherent dataset flaws:

- Testing private functions: Private functions cannot be accessed from test scripts. If no public wrapper exists, the logic cannot be tested.
- Accessing private variables: Unlike public variables, private variables do not expose implicit getter functions. Tests attempting to access these fail without a workaround.
- Deploying abstract contracts: Some tests attempt to instantiate abstract contracts, which is invalid in Solidity.
- Inter-contract interaction failures: Certain contracts require the deployment of a secondary contract to interact with. Although the LLM occasionally attempts to generate such supporting contracts, it never succeeds in producing a functioning setup. Consequently, functions dependent on inter-contract calls remain untested.
- Original contract bugs: As the dataset consists of contracts marked as "potentially vulnerable", some Solidity files contain actual logic errors. For example, one contract deducts funds twice in a single send operation. The AI-generated test logically expects one deduction and one addition, resulting in a failing test that cannot be corrected without altering the original, buggy contract.
- Missing initial balances: Tests often assume the presence of an initial token or currency balance to perform buy/sell operations. If the contract has no constructor or mechanism to initialise this balance, the test fails by design and cannot be salvaged.
- Multi-contract logic within a single file: Some contract files define two or more interacting contracts with tightly coupled logic. These internal dependencies often confuse the AI, and due to their complexity, manual fixes are equally unfeasible.
- Unexplained virtual machine errors: In some cases, calling certain functions triggers invalid VM opcode errors. These issues cannot be traced back to a fixable mistake and are therefore discarded after investigation

These issues, both simple and resolvable, or too complex and hard to fix, shape the final composition of the testbench. While many tests can be salvaged through targeted adjustments, others prove too complex to fix due to fundamental limitations. Either in the LLM's understanding or the structure of the contracts themselves. An important remark is that no modifications are made to the original contracts throughout this process, even when certain changes could have resolved failing tests. This decision is taken to avoid introducing unintended behaviour and to maintain consistency across all evaluations. Despite these constraints, the resulting test suite, after cleanup, offers a diverse foundation for the amplification experiments.

4.5 Testbench Creation 2

During preliminary evaluation of the results on the original test suite (Testbench 1), it became apparent that direct comparison between search-based and LLM-based test amplification techniques was not entirely fair. Specifically, LLM-based techniques gained a disproportionate advantage due to their ability to generate new tests targeting functions not covered at all in the original test suite. In contrast, search-based methods are confined to amplifying existing tests and cannot reach untested functions unless indirectly through existing control flow. This structural imbalance, further discussed in Section 5.5, led to the design of a second, more balanced test suite.

To address this, a new strategy is adopted that aims to level the playing field by amplifying the original testbench. This amplified version is specifically designed to cover as many functions as realistically possible in advance, thereby limiting the structural advantage that LLMs have in testing previously untested functions. Achieving complete function coverage is unattainable due to the technical constraints discussed in Section 4.4.7, but this is not necessarily problematic. The core requirement is that the function coverage should be so that remaining untested functions are untestable in practice, either due to internal errors or inaccessible paths, rather than due to oversight in test construction. This ensures that any function coverage gain by the LLMs is not the result of reaching functions that are overlooked by the base testbench, but rather from untested functions that are invoked along alternative paths within already-tested entry points.

To construct this improved dataset, the existing testbench is used as a foundation. This decision is motivated by the dataset's diversity, as it includes contracts drawn from various domains such as decentralised finance, blockchain-based games, and token exchanges; there is no need for better contracts, just better tests. The aim is to amplify the existing testbench of 923 tests by using LLM-assisted test generation to target functions that were previously left untested.

The prompt used to guide this generation process is as follows: **Here are Solidity contracts 2018-****.sol and their test file named 2018-****-test.js in the hardhat_testing/test directory. Write an extended version of the test class that includes additional unit tests to increase the function coverage of the class under test to 100 percent. Do not create any new files, mock contracts, or use mocking libraries. Focus on testing the existing contract functions with various inputs and scenarios.**

In the augmentation phase of the original testbench, Gemini 2.0 is employed in place of GPT-4, which had been used during the initial testbench generation. This choice is informed by the high number of syntactic errors encountered when using GPT-4, suggesting that it is not the most reliable model for generating syntactically valid tests. In contrast, Gemini 2.0 produces fewer errors overall, making it a more suitable candidate for the task of testbench amplification. The goal remains to expand coverage without introducing new files or mock elements, while maintaining syntactic and semantic correctness across a diverse set of smart contracts.

Additionally, amplification experiments are conducted using various LLMs on the original GPT-4-generated testbench. Interestingly, GPT-4 itself performs the worst in this scenario (see Section 5.2 in the Results chapter where the LLM results are discussed). One plausible explanation for this outcome is that using the same LLM for both generation and amplification reduces diversity in output style and logic, leading to diminishing returns. While this hypothesis requires further investigation, it raises important questions about the impact of stylistic repetition in multi-phase LLM-driven workflows. This topic will be explored in greater depth in Chapter 6, where the threats to validity are discussed.

After the tests are generated, a manual review is conducted to correct invalid outputs and ensure compatibility. This includes identifying and fixing syntactic issues, adjusting incorrect assumptions made by the LLM, and writing additional tests for functions that remain untested even after amplification. This step is crucial, as the LLM-generated tests often require refinement to be fully functional and to align with the correct execution context of the contract. This behaviour was also visible during the generation of the initial testbench.

As a result of this multi-step process, the testbench expands from 923 to 1566 validated test cases. The average function coverage across the dataset reaches 77%, with no individual contract falling below 64%. Some files exceed 90% function coverage, indicating scenarios where only a few functions remain uncovered. These residual gaps typically correspond to functions that are theoretically reachable, such as private functions called only through rare branches, or genuinely untestable due to runtime errors, contract misconfigurations, or other structural issues discussed in the aforementioned section on test evaluation limitations. The dataset's contracts are known to be "maybe vulnerable", which reinforces the plausibility of internal function failures.

By including nearly all reachable functions in the amplified testbench, the evaluation framework prevents LLMs from inflating function coverage scores by simply discovering testable but previously untested functions. This results in a fairer basis for comparison, where both LLMs and search-based methods are evaluated on their ability to navigate and explore already covered functional spaces. Consequently, future measurements of branch and line coverage are less likely to be distorted by baseline discrepancies in function coverage.

Finally, it must be mentioned that certain contracts gained little to no increase in function coverage, during the creation of the second testbench. This hindered the goal of getting the function coverage as high as possible. As mentioned earlier, achieving 100% coverage is usually not possible, but getting to 70-80% is a definitely feasible in many cases. For this reason, any contract that had a function coverage of 60% or lower, was excluded from the testbench. This exclusion criterium is required to ensure enough fairness during amplification with search-based and LLM-based techniques. Because of this, an additional 12 contracts had to be removed from the dataset, leaving a total of 101 contracts. While one can argue that retaining only contracts with 70% or more function coverage would lead to an even fairer comparison, this would require a significant portion of contracts to be excluded, which decreases the representability of the dataset.

Descriptive statistics of the testbench

Much like the first testbench, the newly created testbench should also be a meaningful and representative benchmark for evaluating test amplification techniques. This assessment starts with the distribution of achieved coverage values prior to amplification.

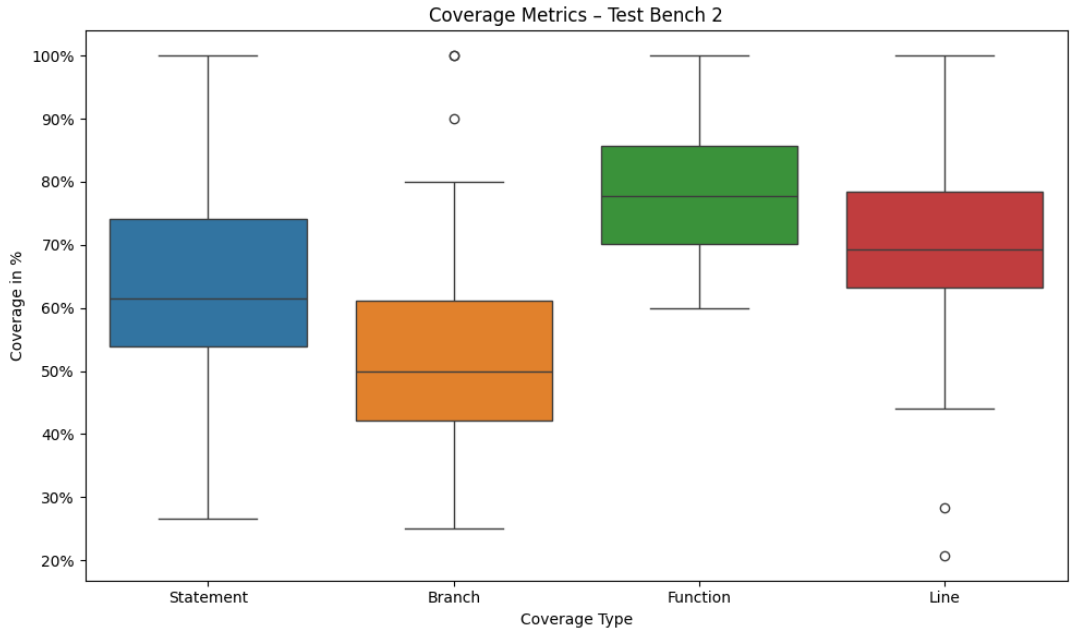
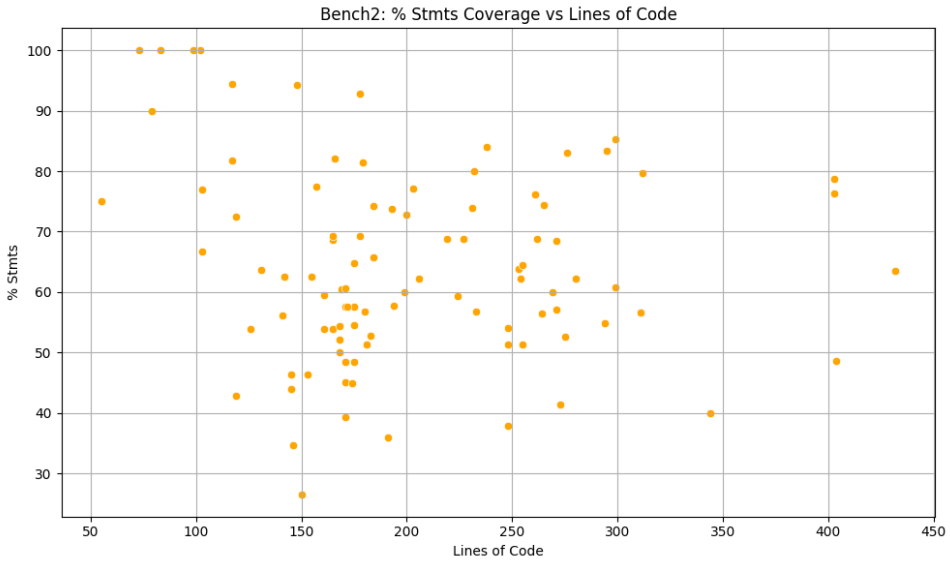


Figure 4.7: Coverage distribution testset 2

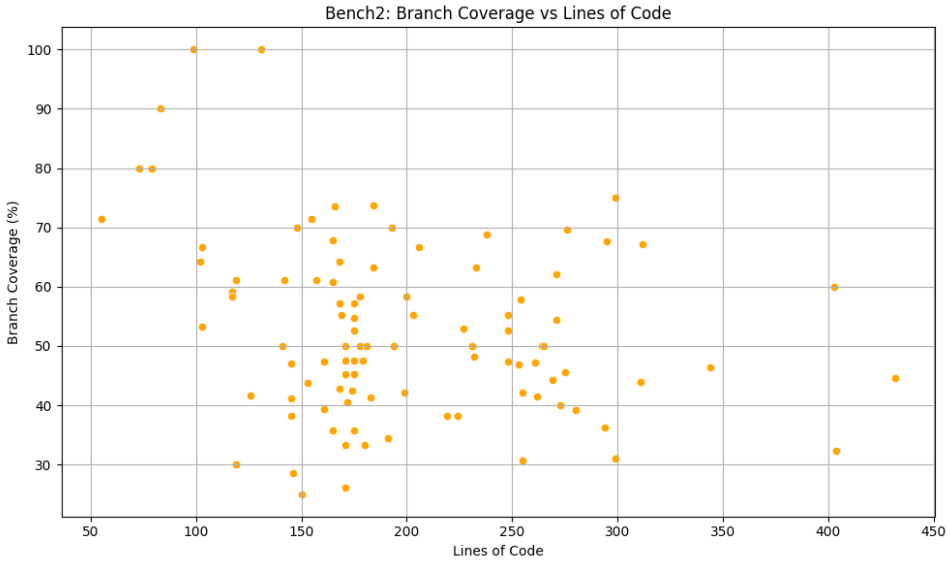
In the box plot in Figure 4.7, it becomes apparent that the achieved coverage by the second test suite still varies substantially. However this time, the coverage metrics on the y-axis start from 25%, showing that the test suite has significantly higher code coverage metrics. This is especially visible in the function coverage, which has a minimum of over 60%. Almost half of the tests achieve 80% or higher, and some reach 100%. Again, the results seem to be well distributed.

Similar to before, two more plots are shown in Figure 4.8. It shows the statement and branch coverage plotted to the lines of code. The remaining plots for the other code coverage metrics can again be found in Appendix A.3. The same statement as before can be made: larger test files

do not systematically achieve higher or lower coverage than smaller ones, making this dataset suited for further analysis.



(a) Statement coverage to LOC 2



(b) Branch coverage to LOC 2

Figure 4.8: All coverage metrics plotted to the Lines Of Code (LOC) 2

4.6 Amplification methods

This section outlines the two automated test amplification strategies evaluated in this thesis: search-based amplification and LLM-based amplification. Both approaches aim to enhance existing test suites for Solidity smart contracts by generating additional test cases, but they differ fundamentally in their way of working. Search-based amplification applies systematic input mutation and guided exploration techniques to evolve tests, while LLM-based amplification leverages large language models to generate new test logic based on code context and existing tests. The following subsections describe the design and implementation of each approach in detail.

4.6.1 LLM-based Test Amplification

The objective of the LLM-based test amplification is to extend existing test suites for Solidity smart contracts by generating additional test cases using large language models. This approach relies on prompt-based amplification, where an LLM is guided to create new tests that cover untested functionality or edge cases. Unlike search-based methods, which rely on input mutation and algorithmic exploration, LLM-based amplification leverages pretrained code understanding and generation capabilities to produce human-like test logic based on the structure of existing contracts and tests.

The aim is to use a consistent prompt template in combination with a specific LLM to generate syntactically correct and semantically meaningful test cases. These amplified tests are intended to complement the original suite by increasing code coverage and exercising more diverse contract behaviour. This section outlines the steps taken to configure, apply, and evaluate the LLM-based amplification approach.

4.6.2 Exploratory Evaluation of Open-Source LLMs for Test Amplification

In the first stage of the LLM-based test amplification process, several open-source language models are evaluated for their potential to generate additional test cases for Solidity smart contracts. The aim is to identify a freely available model that can accept a structured prompt containing both a smart contract and its corresponding test file, and return valid JavaScript test code suitable for use within the Hardhat testing framework.

Some models are accessed through the Hugging Face Inference API [92], which allows cloud-based execution of models without requiring local installation or GPU resources. Specifically, Phi2, StarCoder, Mistral, and Qwen are queried using this hosted interface. Other models, including CodeGen and LLaMA, are executed locally using the transformers library, an open-source Python toolkit developed by Hugging Face that provides a unified interface for loading and running pretrained models on local hardware. Local execution is performed on a machine equipped with an NVIDIA RTX 4090 GPU to accommodate the memory requirements of these models. Hugging Face serves as both a model-sharing platform and a development hub for open machine learning tools. The Hugging Face Model Hub [93] offers thousands of pretrained models across domains (NLP, code generation, vision, etc.), while the transformers library facilitates seamless integration of these models into local or remote workflows. Together, these tools enable flexible experimentation across a variety of model types, making it possible to compare the performance of instruction-tuned, decoder-only models under different deployment settings.

Different models are tested using Hugging Face-hosted APIs and local model loading via the transformers library.

These models include:

- LLaMA-2-7B (Meta)
- Mistral-7B-Instruct (MistralAI)
- Phi-2 (Microsoft)
- Qwen2.5-Coder-3B (Alibaba)
- StarCoder (BigCode)

The selection of open-source large language models for this exploratory evaluation was guided by three key factors: practical availability, potential suitability for code generation and test amplification, and prior adoption in related research. The study by Alshahwan et al. [1], which explores the use of LLaMA-based models for automated unit test improvement at Meta, served as a primary motivation to include LLaMA-2-7B (Meta) as the baseline model for this work.

All selected models are accessible via the Hugging Face Model Hub, either for local execution using the Transformers library or via remote inference through the Hugging Face Inference API. However, as a student researcher operating under Hugging Face's free-tier constraints, access was limited to models available under open-source licenses and permissive usage conditions.

In this setup, LLaMA-2-7B was executed locally using the transformers library, since it is not available via the Hugging Face Inference API without special access approval from Meta. Despite being run on an NVIDIA RTX 4090 GPU, memory usage had to be explicitly limited to avoid runtime instability and crashes during inference introducing further constraints on model performance and input size.

The remaining models Mistral-7B-Instruct (mistralai/Mistral-7B-Instruct) [2], Qwen2.5-Coder-3B (Qwen/Qwen2.5-Coder-3B-Instruct) [3, 4], StarCoder (bigcode/starcoder) [6], and Phi-2 (microsoft/phi-2) [5] were accessed remotely through the Hugging Face Inference API. While these models were chosen for their code generation capabilities, their use via hosted inference endpoints also introduced practical limits in terms of prompt size and output length.

Additionally, CodeGen-6B-mono (Salesforce/codegen-6B-mono) [7] was considered due to its availability on Hugging Face under a permissive license. However, only the mono-language variant primarily trained on Python was accessible for free-tier users. As this model lacks training on JavaScript, it proved incompatible with the smart contract test amplification task, which relies on generating JavaScript-based unit tests.

While the model selection initially emphasised general code generation strength, this setup also reflects real-world constraints encountered by researchers without commercial cloud access. As elaborated in the following sections, many of these models ultimately underperformed in this specialised task, due to a combination of misaligned training data, insufficient reasoning, or limited context windows.

To evaluate the ability of open-source large language models for test amplification in smart contract contexts, a preliminary test was conducted using a minimal setup: a simple Solidity contract and its associated JavaScript test file, each containing approximately 100 tokens. The prompt used for this preliminary test is the following. Anything between curly brackets is replaced by the exact code:

Here is a Solidity smart contract: {contract_code}. Here is the existing JavaScript test file: {test_code}. Write additional unit tests in JavaScript to improve coverage of the Solidity contract.

This served as a baseline to assess syntactic and logical behaviour without hitting model-specific token constraints. Under these conditions, most models returned syntactically valid output. Some even included rudimentary assertions and test logic. However, noticeable differences emerged in terms of language alignment and instruction following. For example, Phi-2 produced short, shallow completions. Mistral-7B-Instruct, Qwen2.5-Coder-3B, and StarCoder generated better-structured JavaScript code but lacked diversity and depth. LLaMA-2-7B occasionally showed promising results but suffered from inconsistencies.

These early observations highlighted that, while capable of basic completions, most models already showed signs of limited reasoning depth and misalignment with the prompt intent even when prompt size was well within their context window.

The actual use case involves significantly larger inputs: contracts from the test suite span approximately 2,000 tokens, and when combined with their associated JavaScript test files, the total prompt size reaches 4,000–5,000 tokens. At this scale, many models become non-viable due to hard context window limits. Table 4.1 shows the token limits for the discussed models.

For smaller-context models, such as Phi-2 and LLaMA-2-7B, the prompt is frequently truncated, resulting in broken, incomplete, or misaligned outputs. Even for models with significantly larger theoretical context windows such as Mistral-7B-Instruct and Qwen2.5-Coder-3B the generated test cases were often semantically weak, repetitive, or overly shallow, failing to introduce meaningful assertions or test logic variation.

Table 4.1: Token limits of various language models [1, 2, 3, 4, 5, 6, 7]

Model	Token Limit
LLaMA-2-7B (HF format)	4096
Mistral-7B-Instruct v0.2	32768
Phi-2	2048
Qwen2.5-Coder-3B	32768
StarCoder	8192

Although the token limitations of many open-source models are known in advance, these models were still included in the exploratory evaluation for several reasons. First, their accessibility via Hugging Face's free-tier API made them a practical starting point for initial testing. Aside from that, part of the goal was to empirically confirm whether these models could still generate meaningful or structurally valid output under realistic test amplification settings. This evaluation helps establish a performance baseline and underscores the gap between open-source LLM capabilities and those of proprietary models, which motivates the switch to more advanced systems in the following sections.

So, evaluating models purely based on token capacity alone is not sufficient. The reasoning depth, instruction tuning, and code synthesis alignment of a model are equally critical. These issues are further compounded by the use of hosted inference endpoints (e.g., Hugging Face's inference API), which may apply stricter practical limits than the models' theoretical capacity especially under free-tier conditions.

Due to these limitations, it becomes clear that high-quality amplification requires access to more capable instruction-tuned models, such as GPT-4, Claude, or Gemini, which are not available via open-source or free-tier APIs. As a result, an alternative setup is required to integrate these models into the workflow through other interfaces. The next subsection discusses the selection and integration of such models using IDE-based solutions.

4.6.3 Enhanced Test Amplification via GitHub Copilot Pro and Commercial LLMs

Following the limitations observed with open-source and Hugging Face-hosted language models, a new approach is adopted to achieve higher-quality test amplification for Solidity smart contracts. Instead of relying on model-specific APIs or local deployments, the strategy shifts toward using GitHub Copilot Pro as a centralised interface for accessing multiple state-of-the-art large language models. This includes GPT, Claude, and Gemini, depending on system configuration and user preferences.

GitHub Copilot Pro offers deep integration into Visual Studio Code, providing intelligent code suggestions, in-place editing, and context-aware completions via Copilot Chat and Copilot Edit. This is the same integration that is used in the test generation (see subsection 4.4.4 about test generation). This could have an impact on some of the results, and will be discussed in Chapter 6 about the threats to validity. However, the features from this setup allow the LLM to access and understand both the Solidity contract and its associated JavaScript test file within the same workspace. By leveraging this integration, the LLM can generate new test cases directly within the test file, maintaining the structure and conventions of the original test suite.

These commercial LLMs are selected based on their publicly reported strengths in instruction-following and code generation. Compared to earlier open-source models accessed via Hugging Face, proprietary models such as those in the GPT, Claude, and Gemini families offer substantial improvements in prompt understanding, generation quality, and contextual reasoning. These improvements are particularly relevant for tasks like unit test generation, where domain-specific knowledge, such as Solidity syntax, contract behaviour, and output precision are critical.

The amplification process is conducted manually for each smart contract. For every contract in the dataset, the associated Solidity file and JavaScript test file are opened, and a tailored

prompt is provided to the LLM. The exact prompts will be discussed later in the relevant sections. In the case of GitHub Copilot Pro, suggestions are generated inline within the test file using the Copilot Edit interface, allowing the amplified tests to be added directly to the existing code.

A key limitation of this approach is its manual nature. As batch processing across multiple files is not currently supported in any of the tools used, the procedure must be repeated individually for all contracts in the dataset. However, tests generated through these LLMs often include assertions, edge case handling, and error conditions that align well with the semantics of the Solidity contracts.

This setup enables test amplification with modern, high-performing LLMs without the need for paid API keys or local hosting. However, to replicate this setup, access to Github Pro is required. While it is freely available for students and researchers at universities, it can't be considered a free tool. Proceeding, by accessing multiple models across different providers, this strategy allows exploratory comparison of output quality between various LLM families. In the results section, it will become apparent how prompts are constructed for each tool, and how differences in test output across models are observed and interpreted. [27, 28, 94, 95]

4.6.4 Experimentation using Pilot Studies

To identify the most effective amplification setup, a two-stage pilot study is first conducted before applying any approach to the full dataset. Given that amplification through Copilot Pro is a manual process each file must be individually opened and processed, it is infeasible to test every possible combination of prompt variant and LLM across all contracts in the dataset. Therefore, the evaluation begins with a small-scale test on a representative subset of 20 contracts. These contracts are amplified using three distinct prompt variants, all inspired by prompt templates proposed in the Meta study by Alshahwan et al. [1], which explored prompt engineering strategies for LLM-based Solidity unit test amplification. The three prompt variants are:

- **Corner Cases**: Designed to guide the model toward generating tests for uncommon or edge-case scenarios.
- **Increase Coverage**: Explicitly asks the model to expand the test suite in order to improve coverage metrics.
- **Combination Prompt**: Merges the instructions of the previous two prompts, requesting both increased coverage and inclusion of corner cases.

Each amplification run is carried out in isolation to prevent cross-contamination of results, and the resulting test files are evaluated using the standard Hardhat and Solidity-Coverage pipeline. The comparison focuses primarily on branch coverage improvements, with secondary metrics such as line and statement coverage also recorded where relevant. The selected contract subset of 20 contracts includes a diverse mix of high-coverage and low-coverage cases, ensuring that the effectiveness of each prompt is assessed across varying baseline conditions.

In the second phase, the same 20-contract subset is used to evaluate the performance of five different LLMs in response to the best prompt. This helps isolate the effect of the model itself while keeping the prompt and evaluation setup constant. The following five LLMs are included in this comparison. These five are all the non-preview models available in Github Copilot Pro, during the time of experimentation. Preview models are also provided, such as GPT-4.1, Claude Sonnet 4, Claude 3.7 Sonnet Thinking, o1 Preview, o4-mini Preview, and Gemini 2.5 Pro, but were untimately avoided due to their unstable performance or longer latency. Preview models are still under evaluation or only introduced shortly before final writing, which makes them unsuitable for systematic benchmarking. Below is a list of the five non-preview models, all of which are used in the comparison:

- ChatGPT O3 Mini (OpenAI): Lightweight, fast, and efficient model for reasoning and code generation.
- ChatGPT 4o (OpenAI): OpenAI's latest flagship model designed for advanced reasoning, coding, and real-time multimodal tasks.
- Gemini 2.0 Flash (Google DeepMind): Lightweight model tuned for speed and long-context tasks.
- Claude 3.5 (Anthropic): Mid-tier model from the Claude 3 series, aimed at efficient general-purpose and coding tasks.
- Claude 3.7 Sonnet (Anthropic): Latest high-end Claude model optimised for long-context reasoning and code understanding.

This experiment investigates both the quantitative coverage improvements and qualitative output characteristics of each model. While all models receive the same prompt and contract/test context, significant differences in test logic, stability, and adherence to formatting and syntax constraints are expected. [96, 97, 98, 99, 100, 101]

4.6.5 Search-Based Test Amplification

The objective of search-based test amplification is to enhance the effectiveness of the initial test suite by generating new test cases that improve its coverage and fault-detection capabilities. Rather than replacing the existing tests, this approach aims to build upon them by systematically introducing small modifications, such as input mutations, assertion insertion, or condition exploration, to discover previously untested behaviours. This method operates under the assumption that existing tests provide a solid foundation but do not yet explore all relevant execution paths or edge cases. The goal is therefore to automatically evolve these tests into more variants, thereby increasing the overall quality of the testing process without requiring manual intervention.

4.6.6 Setup for the Search-Based Test Amplification

To perform test amplification on the manually created testbench, three distinct algorithms are implemented. Each of these builds upon the same basic mechanism, mutating existing test cases, but differs in how mutations are selected and combined to explore new program behaviour. The aim is to cover more branches of the tested contracts by modifying existing tests in increasingly intelligent ways. The first two algorithms (random and guided search) operate in a stateless manner, meaning that each amplification iteration always starts from the original, unmodified tests. In contrast, the third algorithm, genetic search, builds new tests incrementally across generations. It introduces the concept of "evolution" through selection, mutation, and crossover, with each generation depending on previously evolved test cases. What follows is a detailed overview of each algorithm and how it is applied in the context of this experiment.

The first of these is a basic random search, serving as a simple yet effective baseline. This algorithm operates by identifying the data types of variables used in the test cases, such as uint, int, bool, address, and string, and applying random mutations to randomly picked lines of code within the test. These mutations typically involve substituting a value with another of the same type. For instance, an integer like 100 might be replaced with a random number such as 42672. Alternatively, mutations may incorporate common edge cases, including zero, boundary values (maximum/minimum for integer types), negative numbers, or even syntactically valid but semantically different values. This naive approach does not rely on contract semantics, making it easily applicable but potentially less effective for complex behaviours.

The second algorithm builds upon the basic random search, introducing a more guided approach to mutation selection. While still fundamentally random in nature, this variant incorporates some heuristics to increase the likelihood of discovering meaningful variations in behaviour.

Rather than selecting lines for mutation entirely at random, this version prioritises modifying parts of the test that are more likely to affect control flow, such as function calls with multiple arguments, while largely avoiding lines that are unlikely to yield new coverage (e.g. simple getters, which typically succeed regardless of input and do not influence branching behaviour).

The mutation values themselves are also chosen with more intent. For instance, integer arguments are more likely to be mutated to edge cases such as 0, boundary values, or negative numbers, though purely random values remain possible with reduced probability. This selective bias aims to improve branch exploration while retaining the generality and simplicity of a randomised method.

Importantly, this approach still treats the contract under test as a black box: it does not parse or analyse the Solidity source itself. This design choice ensures that the algorithm remains focused on test-level amplification, independent of the contract's internal structure, which aligns with the goal of amplifying existing tests without introducing external dependencies.

The third and most sophisticated algorithm is a genetic search, designed to improve code coverage through evolutionary principles. Unlike the previous methods, this algorithm not only uses mutation, as seen in the random search, but also introduces two additional operations: selection and crossover.

Genetic search begins with an initial population of test cases (generation 0), consisting of the original tests. From this base, new generations are iteratively created. In each generation, selected test cases are mutated and/or recombined through crossover to produce new candidates. The selection step retains only the most promising test cases, based on fitness criteria such as branch coverage or execution success.

Crossover refers to the combination of parts from two different test cases to produce a new one. In this context, this means taking lines from two different tests, potentially from different generations, and merging them into a single test script. For example, one part of a test may set up a specific contract state, while another part from a different test might do other operations. By combining both, it becomes possible to generate new behaviour that exercises previously unexplored paths in the contract, resulting in a more complex sequence of interactions. Crossover is not limited to adjacent generations; it can occur between any generations (e.g., between generation 2 and generation 6), allowing the algorithm to recombine effective strategies discovered at different stages of the search process.

This iterative approach allows the algorithm to gradually refine the test suite, balancing exploration (through random mutation and crossover) with exploitation (by selecting high-coverage test cases), aiming for broader and deeper test coverage than the previous methods.

In the genetic search approach, selection plays a crucial role in determining which test cases are preserved and carried forward into the next generation. The main goal is to incrementally improve coverage, so only test cases that provide measurable value are retained. First and foremost, only passing tests are considered. Failed tests are excluded from further processing, as they are not useful for amplifying functional behaviour and may introduce noise into the evolutionary process.

Among the successful tests, each one is evaluated based on whether it contributes new coverage. This is done by checking if a test, when added to the current set, results in the execution of previously uncovered branches or paths. Tests that do not improve coverage are discarded, while those that do are flagged as valuable. This ensures that each generation builds upon meaningful progress and avoids redundancy. The process mimics a fitness-based selection mechanism, tailored specifically to line and branch coverage in the Solidity smart contract environment. [29, 30, 31]

Finally, although the setup includes dynamic execution to measure coverage (via Solidity-Coverage), this runtime information is not used to guide the test generation process. The search strategies explored in this thesis are therefore considered static. More advanced techniques such as concolic testing [102] or dynamic feedback-driven fuzzing [103], which combine runtime insights

with symbolic execution or search steering, are beyond the scope of this work and are discussed in related research.

4.6.7 Selection in Practice: Coverage-Based Filtering with Batching

In theory, selection in a genetic search algorithm is relatively simple: only valid test cases that contribute positively to overall code coverage should be retained. In practice, however, when working with Solidity contracts and the Hardhat framework, several limitations arise that complicate this process.

One key limitation is that the coverage report does not indicate which specific tests contribute to the coverage increase. While it is possible to observe that a group of tests improves overall coverage, the exact tests responsible for this improvement remain unknown. A solution would involve executing each new test individually, analysing its impact on coverage, and retaining only those that lead to an increase. However, this method is not feasible for larger datasets. With 117 contract files and an average of 9 to 10 tests per contract, such a detailed evaluation would require an impractical amount of time.

To manage this challenge more efficiently, a batching approach was used. Typically, each test case from generation zero can undergo a mutation when growing the population towards generation one. Additionally, several new tests are created through crossover, depending on compatibility with the crossover operation. After removing failing tests, the total number of tests per contract tends to roughly double with each generation.

The newly generated tests are divided into four equal parts (approximately 25% each) per contract. These parts are then tested separately in four coverage analysis runs. If one of the parts leads to a coverage increase, even if only a single test within it is responsible, the entire 25% batch is retained. This strategy reduces the number of required coverage runs to just four per generation as it is possible to select a batch for each contract for one coverage run, allowing the coverage analysis to run in parallel for all contracts, making it significantly more time-efficient than testing each case individually.

Although this method may retain tests that do not contribute uniquely to coverage, the trade-off is considered acceptable. Especially in later generations, where tests are often small variations of earlier ones, it becomes less likely that any single test adds meaningful new coverage. In such cases, entire 25% batches are frequently discarded, increasing overall efficiency. In rare instances, all tests in a batch must be retained due to one valuable test being included, but this is a minor drawback within the broader strategy.

The choice to divide the newly generated tests into four equal batches (25% each) was based on a balance between filtering precision and practical efficiency. While alternatives such as $1/3$ or $1/5$ are possible, they each present drawbacks. A batch size of 33% would result in fewer filtering steps but larger batches, increasing the likelihood of retaining redundant tests. On the other hand, smaller batches, such as $1/5$ or $1/6$, would indeed reduce the chance of redundancy within each retained group. However, this would require more coverage runs per generation. Since the number of generations cannot be reliably predicted in advance and depends on how well the exploration performs on a given dataset, as suggested by Muhammad, A., et al. [104], even a single additional coverage run per generation accumulates quickly. For example, increasing from four to five batches would mean one extra run per generation, which, over many generations, results in a significant increase in execution time and manual effort. Therefore, 25% was chosen as a practical compromise, offering enough granularity to discard low-value tests while keeping the overall evaluation process efficient and scalable.

4.6.8 Implementation of Mutation Operator

The mutation operator plays a central role in exploring the input space of test cases. The main focus here lies on input mutations, where the goal is to alter the parameters provided to function calls within existing tests in order to activate new code paths and increase coverage. These same

mutations are used across random search, guided search, and genetic search algorithms, with the key difference being how mutated inputs are selected and retained. Essentially, in random search, all mutations can be selected with the same probability, while guided and genetic search tend to favour edge case mutations, like zero, maxint etc.

Input mutations are effective in Solidity because many contracts contain conditionals, modifiers, and function-level permissions that depend heavily on specific input values (e.g., address roles, uint values triggering edge cases, or bytes arrays influencing logic). Below is a list of the most relevant mutation types, all of which were used in the implementation of the mutation operator:

- **Numeric Value Perturbation**: Modifying integer or unsigned integer values by adding, subtracting, doubling, halving, or replacing them with edge-case constants (e.g., 0, 1, type(uint256).max).
- **Boolean Flipping**: Inverting boolean inputs, which often toggle important branches, such as enabling/disabling paths.
- **Address Substitution**: Replacing address arguments with random accounts, known privileged users (like owner), or edge-case addresses (0x0, 0xffff...), to influence role-based checks.
- **Bytes and String Mutation**: Altering strings or byte arrays by inserting special characters, truncating, or extending them. This can uncover parsing errors, revert conditions, or path-sensitive behaviours.
- **Array Manipulation**: Changing the length or contents of input arrays, often to trigger loops or indexing conditions inside contract logic.

In addition to input mutations, this implementation also performs assertion addition. All reasoning for assertion placement is done via static analysis only; no runtime behaviour is observed or inferred. While input mutations aim to increase structural coverage (e.g., branches, statements, functions), assertion addition targets semantic quality: the goal is to make tests not only reach new code but also verify that the observed behaviour aligns with expected properties. In practice, this involves appending new assert, expectRevert, or expectEmit statements after contract interactions. These assertions check state variable changes via direct reads or getter calls, reverts or emitted events, or in some rare cases, return values of function calls. The latter is less useful as functions in Solidity contracts often don't return anything, and simply alter the internal state of the contract.

In the case of random search, these assertions are often added in a random manner, not looking too much at the broader context, but rather by looking at already present assertions and aligning with them. For the guided search and genetic search, a heuristics-based approach is taken. For example, once a literal is mutated (e.g., replacing 100 with 0), the algorithm attempts to identify and adjust other values in the test that depend on the original value. Several types of heuristic correlations are applied:

- **Direct correlation**: If an assertion expects the exact value passed into a method where a mutation occurred, it is updated accordingly.
- **Subtraction correlation**: If a value is derived from subtracting a transferred amount from an initial balance, the result is recalculated and updated in the assertion.
- **Addition correlation**: If a value increases due to a deposit or mint operation, the new expected outcome is adjusted.

These correlations are probabilistically applied, roughly three-quarters of the time, to avoid overfitting and allow room for mutation exploration. If a correlation hypothesis leads to a failing test, that specific mapping is discarded in future generations. Such correlation mappings are identified before any operation occurs, and for many of them, no assertion is present yet. So not only are existing assertions updated accordingly, this approach also allows for the input of new assertions, based on the correlations between variables. Assertion addition is particularly valuable because it allows for the detection of more faults. [105, 46]

4.6.9 Stopping Criteria for Random and Guided Search

The random and guided search algorithms are implemented as an iterative process where, at each iteration, a single mutation is applied to an existing test file. Unlike generation-based evolutionary methods, this approach does not maintain or evaluate a population of candidate tests in each cycle. Instead, the process proceeds sequentially, applying one mutation at a time without prior knowledge of its impact on coverage. The only criterion for retaining a mutated test is that it passes execution; tests that fail are immediately discarded. Stopping criteria are therefore crucial to determine when the search should terminate, balancing thorough exploration with computational feasibility. Common stopping conditions include a fixed maximum number of iterations or a plateau in improvements, such as halting after a specified number of consecutive iterations yield no passing mutants [106]. These criteria ensure the search is bounded and prevent unnecessary computation once further meaningful progress becomes unlikely.

In Section 4.6.7, it was apparent that evaluating whether a single test case contributes to coverage is challenging with the Solity-Coverage tool in the Hardhat framework. Therefore, the selected stopping criterium is a fixed number of iterations, rather than a plateau in improvements. To ensure a fair and efficient allocation of mutation attempts across test files, the stopping criterion was not based on the number of test functions per file but on the potential mutation surface. More so, the maximum number of iterations was defined as a linear function of the total number of mutatable lines L, i.e., lines in the test file that perform operational logic rather than assertions or comments. This approach acknowledges that test files often differ in granularity: some functions may contain only a handful of statements that can meaningfully be mutated. By scaling the iteration budget in proportion to the number of mutatable lines, rather than purely by function count, the search process remains sensitive to the actual opportunities for test amplification, avoiding both under- and over-processing of files with varying structural complexity.

The number of iterations N for a given test file is defined as $N = a \cdot L$, where L represents the number of mutatable lines, a is the density factor. The number of tests per file ranges from two to fifteen. The decision to fix $a = 3$ is made. It is based on the observation that most operational lines in test files encapsulate only one to three functional actions that can be mutated, and that around three distinct mutation attempts per such line offer a practical balance between diversity and efficiency. This strategy enables a consistent mutation effort across test files of varying complexity while maintaining computational feasibility, especially for the test files with fifteen functions that contain many lines within them. Finally, where a mutation occurs and what this mutation will be, is chosen at random. [106]

4.6.10 Stopping Criteria for Genetic Search

In genetic algorithms, stopping criteria are essential to prevent excessive computation and to ensure practical convergence. While a wide variety of termination conditions are theoretically available, not all are suitable for every application. In the context of test amplification for Solidity smart contracts, certain common conditions, such as fitness thresholds, known optimal solutions, or fixed time limits, are either inapplicable or insufficient. Instead, stopping conditions must be adapted to the realities of incremental coverage, test redundancy, and practical evaluation

constraints. Especially test redundancy must be taken into account, as the batch filtering based selection will cause a lot of redundant or duplicate tests.

Coverage Convergence Across Generations

The primary stopping condition adopted in this work is based on stagnation in coverage improvement. Since coverage is used as the core fitness signal, generations are expected to progressively increase overall code coverage. However, in practice, the rate of improvement often diminishes after a certain point. Once multiple consecutive generations fail to yield meaningful gains (e.g., an increase of less than 0.25% branch coverage over the previous generation), the algorithm is considered to have converged. This reflects a plateau in the search space where further mutations or crossovers tend to produce even more redundant or low-impact tests.

This form of stagnation-based stopping is preferable over fixed target coverage or absolute fitness thresholds, both of which are problematic in Solidity testing. Due to unreachable/broken functions, legacy code, or external dependencies, the theoretical maximum coverage is unknown and may not be achievable. Thus, setting a predefined target is infeasible, and progress must instead be evaluated relatively. [107, 69]

Evaluation Saturation and Operational Cost

A second, equally important stopping criterion is based on operational overhead. Each generation requires a full set of coverage evaluations across four distinct test batches per contract, in order to identify and retain only those segments that contribute to meaningful coverage gains. While this batching strategy significantly reduces the number of individual test evaluations, it still incurs non-trivial computational and time costs. In cases where this evaluation process becomes saturated, for example when batch execution and filtering begin to exceed a practical time threshold (set to approximately two hours of manual-equivalent effort per generation), the search is halted. This scenario typically occurs when the population has grown large and new individuals offer limited novel contribution, yet still require full evaluation. At that point, the marginal utility of additional generations is outweighed by the operational burden.

Other common stopping criteria not used in the study

- **Diversity collapse and population redundancy**: Although not strictly implemented in the current framework, an additional theoretical stopping condition could involve population diversity. In genetic algorithms, a collapse in diversity, where the population converges to highly similar or nearly identical individuals, often signals that the search space has been fully explored within the given parameters. Diversity can be estimated based on input similarity (e.g., structural features or token overlap between test cases), and if it falls below a predefined threshold, the algorithm may terminate. However, such a strategy requires additional tooling to measure similarity between Solidity tests, which falls outside of the scope of the thesis, and was not implemented. Still, it remains a viable future enhancement for scenarios where test variety is critical. [107]
- **Fixed generation cap**: While a fixed maximum number of generations is a common and simple termination condition in many genetic setups, it was consciously not employed here. Given the variability in code complexity across contracts, a fixed cap could terminate the algorithm prematurely for harder cases or extend it unnecessarily for simpler ones. Instead, stopping is driven by empirical signs of stagnation or operational inefficiency, which better reflect the real progress of the amplification process. [107]

4.6.11 Implementation of Search-Based Techniques

The following section will discuss the general implementation of all search-based techniques. The goal is to take the name and location of a testfile as input, and output an entirely new file, which is both structurally and semantically correct, and contains both the old tests, as well as new, amplified test cases.

Parsing and Internal Representation

Each Solidity test file is first parsed and preprocessed. This preprocessing step removes elements such as whitespace and comments, yielding a clean, line-based representation of the code. Each function in the test file is stored as a list of its instructions, with the entire file represented as a list of such function lists. This intermediate representation allows the mutation algorithm to inspect and manipulate test logic at the level of individual function bodies while preserving their structural context.

So essentially, the representation used is an instruction list, rather than an abstract syntax tree (AST). Both have their pros and cons but here are a few that justify the use of instruction lists, over ASTs. Instruction lists offer a more direct representation, which aligns naturally with the sequential nature of test cases. Since most Solidity tests consist of ordered function calls without deeply nested control structures, a flat list of instructions is sufficient for capturing their semantics. This simplicity reduces the overhead of parsing and structural validation required by AST-based approaches. Moreover, manipulating linear instruction sequences, such as swapping, deleting, or injecting function calls, is more straightforward than performing subtree operations on ASTs, which often require type compatibility and structural integrity checks. While ASTs provide finer-grained control and safety in complex transformations, they are not strictly necessary in this context, making instruction lists a more pragmatic choice. [69, 108]

Amplifying test cases

With the intermediate representation in place, the amplification algorithms can begin processing the existing test cases. For the random search, the process is straightforward: mutations are applied to the tests, as described in Section 4.6.8, and repeated until the stopping criteria are met. The guided search follows a similar pattern but incorporates a lightweight heuristic to direct the mutation process. This often results in fewer generated tests compared to random search, as not every mutatable line is explored to the same extent. Finally, the genetic search algorithm follows a fixed sequence of operations: after an initial population is formed (generation 0), each new generation proceeds with selection (skipped for the initial tests), followed by crossover and then mutation. The resulting test cases are executed, after which the failing test cases are removed, and the remaining ones are evaluated based on coverage. Then, the stopping conditions are checked. If they are not met, the process repeats, discarding any failing tests before moving on to the next generation.

Test Regeneration and Failure Filtering

After all test functions are mutated and adjusted, the test file is reconstructed with uniquely named test cases (e.g., 'test 1', 'test 2, etc.). This regeneration step occurs not only at the end of the amplification process but also after each generation in the genetic algorithm to ensure that test execution and evaluation can proceed incrementally. Once the updated test file is executed using the Hardhat testing framework, the output is parsed to identify failing test cases. These failures may result from invalid input states, unmet preconditions, or syntactic inconsistencies introduced during mutation. To prevent such failures from interfering with the coverage measurement, a script automatically marks the failing tests with a 'skip' annotation. This filtering step is essential, as only passing tests contribute meaningfully to code coverage metrics.

4.6.12 Hybrid Models

In addition to evaluating the standalone performance of search-based methods and LLMs, two hybrid strategies are considered. These hybrid configurations aim to incrementally build the test suite by applying both techniques in sequence. The test suite generated by the first method serves as the input for the second, allowing it to amplify or refine areas that may have been missed. This layered approach enables each method to contribute its strengths: semantic reasoning in the case of LLMs, and structural exploration in the case of search-based methods. By design, the hybrid configurations cannot perform worse than the technique used first, since they extend and enrich its output. This setup creates a structural advantage for hybrid methods and ensures that, in theory, their coverage results match or exceed those of the individual techniques involved.

The metrics used to evaluate these hybrid models, are the same metrics discussed in Section 4.2. Using the same metrics for hybrid models that were also used for standalone models makes sense, as it allows for direct comparison between the two. After all, the goal of creating these hybrid models, is to assess whether it is worth to use them, over standalone techniques. Therefore, a thorough analysis is required between standalone techniques and hybrid models, which is only possible if they are fairly evaluated using the same set of metrics.

Chapter 5

Results / Evaluation

This section presents the results of the experiments conducted to compare different automated test amplification strategies for Solidity smart contracts. The results focus on measuring improvements in code coverage and fault detection achieved through search-based and LLM-based approaches.

5.1 Baseline performance

The evaluation begins with an assessment of the constructed baseline testbench, which includes 113 Solidity contracts. These were selected to represent a variety of real-world contract structure, and each contract varies in complexity, ranging from very short and simple to very long and complex. This ensures that the amplification strategies could be evaluated across diverse testing scenarios. Each contract is accompanied by an initial test suite designed to provide basic functional validation.

To understand the effectiveness of this baseline test suite, four standard coverage metrics are measured: statement coverage, branch coverage, function coverage, and line coverage. These metrics reflect different levels of complexity. Table 5.1 summarises the average values across all 113 contracts.

As shown, the function and line coverage values are relatively higher, indicating that the initial test cases manage to exercise a reasonable portion of the declared functions and source lines. However, the comparatively low branch coverage (32.20%) suggests limited exploration of conditional paths within the contracts. This highlights the potential benefit of amplification techniques, particularly those aimed at increasing control-flow depth in the test suite.

While these averages provide a general overview, the actual coverage values vary considerably between contracts. This was already discussed in Section 4.4.5 and Section 4.4.6 about the evaluation and finalisation of the testbench. Some contracts already exhibit high coverage across multiple metrics, while others show significant gaps. This variability ensures that the test amplification methods are evaluated across a wide spectrum of baseline quality, from well-tested contracts to those with clear room for improvement.

Together, the coverage results establish a representative baseline for evaluating the impact of test amplification. The variation in baseline performance across contracts highlights the importance of amplification methods that can adapt to differing levels of initial test quality. While some contracts already achieve relatively high coverage, others leave substantial room for improvement. This variability underscores the need for techniques capable of enhancing under-tested areas effectively.

This baseline now serves as the foundation for comparing the effectiveness of various search-based and LLM-based amplification techniques in the following sections.

Table 5.1: Baseline code coverage metrics

Coverage Metric	Average (%)
Statement Coverage	45.59
Branch Coverage	32.20
Function Coverage	58.50
Line Coverage	52.99

5.2 LLM-based amplification

5.2.1 First amplification prompt

To perform the test amplification, a similar prompting strategy is applied as in the test generation. The goal in this phase is to extend existing test files with additional unit tests that increase code coverage, specifically branch coverage as this is mostly related to edge cases, without altering the original logic or test structure.

The prompt used for this task is as follows:
"Here is a Solidity contract 2018-***.sol and its test file named 2018-*****-test.js in the hardhat_testing/test directory. Add additional unit tests that will increase the test coverage of the contract. Do not modify or remove any existing tests. Only append new tests that follow the existing test style. Never use .deployed()—this will cause a TypeError unless the contract is compiled with Solidity 0.6.x or higher. Always use ethers.parseEther. Do not use .to.be.revertedWith(); instead, use .to.be.reverted for consistency."**

This prompt is designed to reflect the structural and technical constraints of the Hardhat testing environment. It includes specific instructions to avoid known pitfalls, such as .deployed() issues with older compiler versions, and enforces consistent syntax usage for error assertions and Ether value parsing.

The first round of amplification using this prompt is executed with ChatGPT-4o via GitHub Copilot Pro. Across the full dataset, this initial approach yields an average branch coverage improvement of approximately 7%. While this represents a clear gain compared to the baseline test suites, the improvement is lower than initially anticipated. In practice, several limitations emerge. Despite being explicitly instructed to avoid it, the model occasionally includes calls to .deployed(), which can lead to runtime errors for contracts not compiled with Solidity 0.6.x or higher. This can lead to the failure of an entire testfile, rather than just one test within a file, because the contract under test will be incorrectly deployed. Additionally, certain outputs use the outdated syntax ethers.utils.parseEther instead of the required ethers.parseEther, indicating that prompt adherence is not fully reliable.

Another issue arises when the model attempts to generate tests that rely on the creation of mock contracts (or any external contract really), suggesting the introduction of new .sol files not present in the original codebase. While these tests may be valid in a broader testing context, they are excluded from this study to preserve consistency and ensure a fair comparison with the search-based amplification techniques, which are restricted to operating within the boundaries of the original contract and test file. It's worth noting that the prompt explicitly states to write additional unit tests, but ignores this statement by writing mock contracts anyway. It tries to test the interactions between the original contract, and the newly created mock contract, which would fall under *integration tests*, rather than unit tests.

These findings underscore the importance of prompt clarity, manual validation, and controlled evaluation criteria when applying LLMs for test amplification. They highlight that amplification success depends not only on the language model itself, but also on the structure of the existing test suite and the semantic complexity of the smart contract. In response to these limitations, further experimentation is undertaken to refine the prompt and explore alternative LLMs avail-

able through GitHub Copilot Pro, including models from the Claude and Gemini families. The goal is to determine which combinations of prompts and models yield the highest improvements in test coverage.

5.2.2 Small-Scale Evaluation Based on Prompt Variants

After establishing a baseline using the initial amplification prompt, a series of small-scale experiments are conducted to investigate how different prompt formulations influence the quality of the generated tests and the resulting coverage improvements. For the small-scale evaluation, 20 contracts (approximately 18% of the dataset) were selected from the full collection of 113 smart contracts. The contracts are sourced from a variety of projects and are not ordered in any systematic way (e.g., not by size, complexity, or origin). As such, selecting the first 10 and the last 10 contracts ensured a practical division of labour, while still approximating a random sample due to the absence of internal structure in the list.

Although a stratified selection based on specific metrics such as cyclomatic complexity could increase representativeness, the chosen method was sufficient for the study's exploratory goal. The inclusion of contracts from different positions in the dataset, and thus from potentially different projects or code styles, helped to cover a reasonably diverse subset of contracts. For each of these 20 contracts, test amplification is performed using three prompt variations derived from the Meta paper [1]:

- **Corner Cases**: "Here is a Solidity contract 2018-*****.sol and its test file named 2018-*****-test.js in the hardhat_testing/test directory. Write an extended version of the test class that includes additional tests to cover some extra corner cases."
- **Increase Coverage**: "Here is a Solidity contract 2018-*****.sol and its test file named 2018-*****-test.js in the hardhat_testing/test directory. Write an extended version of the test class that includes additional unit tests that will increase the test coverage of the class under test."
- **Combination Prompt**: "Here is a Solidity contract 2018-*****.sol and its test file named 2018-*****-test.js in the hardhat_testing/test directory. Write an extended version of the test class that includes additional unit tests that will cover corner cases missed by the original and will increase the test coverage of the class under test."

5.2.3 Results different prompts

Table 5.2 below summarises the results of the small-scale prompt evaluation, applied to the subset of 20 contracts. Four coverage metrics are considered: statements, branches, functions, and lines. Along these four metrics is an additional metric: the pass/fail test case count. Each amplification round is conducted using ChatGPT-4o with a different prompt derived from the Meta paper, as previously described. The final reported score is the mean of these coverages of each contract. The number of passing and failing test cases is obtained by summing the total across all 20 amplified test files per prompt.

The "Increase Coverage" prompt yields the best performance across all coverage metrics, significantly improving over the baseline and outperforming both the "Corner Cases" prompt

Table 5.2: Coverage results for different testing strategies

Metric	Original	Corner Cases	Increase Coverage	Combination
Statements (%)	60.85	66.38	72.77	68.96
Branch (%)	42.48	52.43	58.75	56.715
Functions (%)	72.71	77.58	82.82	78.44
Lines (%)	66.62	71.58	77.423	73.99
Passing/Failing	165/0	224/100	261/102	237/141

and the combination prompt. On average, it results in a +17% absolute gain in branch coverage over the original test suites (from 42.48% to 58.75%), exceeding the 7% gain achieved with the initial prompt used in earlier experiments. In terms of behaviour, each prompt exhibits distinct characteristics.

The **first prompt**, 'Corner Cases', tends to focus on formatting adjustments (whitespace, spacing) and occasionally uses ethers.utils.parseEther, despite prompt guidance to avoid it.

The **second prompt**, 'Increase Coverage' generates more extensive test additions and tends to introduce mock contracts and even contract edits, sometimes detecting potential vulnerabilities. While it introduces more test files, it avoids .deployed() usage errors and generally respects Solidity conventions.

The **third prompt**, The 'Combination Prompt' results in the largest test outputs, sometimes modifying comments and spacing. It generates fewer external files, making it faster to integrate, but also introduces the highest number of failing tests, suggesting lower test stability.

Despite the additional assistance provided by the more complex prompts, the results suggest that simpler and more direct instructions yield better coverage improvements and fewer compatibility issues. The "Increase Coverage" prompt stands out for its balance of test quality, correctness, and measurable impact on test suite effectiveness, and is selected for further use in broader amplification experiments.

5.2.4 Small-Scale Evaluation Based on different LLMs

After identifying the most effective prompt in the previous experiment, the "Increase Coverage" variant, a new set of tests is conducted to evaluate how different large language models respond to the same prompt under identical conditions. This phase focuses on understanding the extent to which the choice of LLM impacts the effectiveness of test amplification.

While the prompt selection was based on a small-scale study using GPT-4, it's important to consider that the effectiveness of a prompt may vary between models. Different LLMs interpret instructions in subtly different ways, which means that a prompt optimised for one model may not yield optimal results for another. Ideally, each model would be evaluated with all prompt variants to determine its individual best-performing prompt. However, due to time constraints of running a full evaluation across all models and prompt combinations on the entire dataset, the decision is made to first identify the best prompt using GPT-4 and then apply that single prompt consistently across all LLMs in the larger study. While this introduces a potential bias, GPT-4 can be considered to be a strong reference model due to its high-quality completions [109], and the selected prompt ("Increase Coverage") was chosen because of its generality and simplicity.

In the section below, the results of each of the five LLMs that were selected in Section 4.6.4 will be analysed and compared to each other.

5.2.5 Results of different LLMs

Table 5.3 below presents the results of test amplification using the best-performing prompt (the "Increase Coverage" prompt) across five different LLMs available via GitHub Copilot Pro. All models are evaluated on the subset of 20 contracts, which is a mix of high- and low-coverage cases, using standard coverage metrics: statement, branch, function, and line, along with the total number of passing and failing test cases. The results are computed using the same method as in the prompt evaluation: for each model, Solidity-Coverage output is collected separately for the contracts and tests, and the final value for each metric is the mean of these outputs. The total number of passing and failing tests is obtained by summing the test outcomes across all 20 amplified test files. At times, certain models tend to create mock files to test interactions between contracts. Such files are immediately discarded as the focus is on writing unit tests, not on integration tests.

Claude 3.7 Sonnet consistently achieves the highest performance across all metrics. It yields a branch coverage of 76.27%, which constitutes a 33.79 percentage point increase over the

original test suites. Similar improvements are observed for statement, function, and line coverage, reaching 83.78%, 89.18%, and 86.07% respectively. This model also produces the highest total number of passing tests (421), reflecting both its comprehensive test generation and its ability to maintain syntactic correctness. During amplification, Claude 3.7 generates extensive test files on average around 160 lines of extra test code per contract. It handles Solidity-specific syntax well, introduces minimal versioning issues, making the test files structurally valid. Its success extends to both high- and low-coverage contracts, although in cases with minimal initial coverage, it tends to generate longer and more redundant tests. Nevertheless, its ability to reason over complex contracts while maintaining coherence makes it the most capable model in this study.

Claude 3.5 Sonnet also performs strongly. It reaches 85.7% function coverage and 80.09% line coverage while maintaining a relatively low number of failing tests. Like Claude 3.7, it avoids versioning and syntax errors, and the use of additional mocks. The generated tests are generally smaller and more controlled than those from Claude 3.7, which results in slightly lower coverage, but more maintainable test additions. Claude 3.5 struggles more when faced with contracts that require reasoning over deeply nested conditions or minimal baseline test coverage but remains a reliable performer.

Gemini 2.0 Flash, although designed for fast inference, performs competitively with 61.2% branch coverage and 302 passing tests. Its amplification runs frequently include test logic for contracts or mocks that are not present in the original project. While this wasn't a problem for either version of the Claude LLMs, here, it results in test failures due to undefined tokens or incorrect assumptions about available dependencies. The model also tends to generate version mismatches and errors around placeholder addresses like addressZero, where it tries to import these constants, but from the incorrect libraries. While its performance is statistically close to that of Claude 3.5, its reliability is lower, especially when stricter test suite structure constraints are required.

GPT-4o achieves a branch coverage of 58.75%, with overall solid amplification capabilities. However, it produces the highest number of failing test cases (102), suggesting overgeneration or noncompliance with framework-specific limitations. In low-coverage contracts where only minimal original tests are available, GPT-4o frequently fails to introduce meaningful additional coverage. In these cases, the model tends to generate large blocks of redundant tests. Additionally, it is important to consider that GPT-4o is also used during the original test generation phase earlier in this thesis. This may possibly reduce its ability to identify new logic paths during amplification, as it may reproduce or slightly vary its original outputs rather than generating structurally novel tests. This will be further investigated.

ChatGPT O3 Mini records the lowest amplification gains among the evaluated models, with the smallest overall coverage increases. Despite this, it introduces the fewest failing tests (40) and produces good test additions. It avoids complex test structures and never generates mock files, like Gemini 2.0, which helps it maintain a higher success rate, especially in contracts with already reasonable baseline coverage. However, it often fails to amplify low-coverage contracts unless the prompt is manually reworded or prompted multiple times. Its output tends to exclude necessary describe blocks, which clearly limits its functional depth. This makes O3 Mini best suited for fast, simple amplification where error minimisation is prioritised, but less effective for generating substantial test improvements.

Table 5.3: Coverage comparison across different models

Metric	Original	O3 mini	GPT-4o	Gemini 2.0 Flash	Claude 3.5	Claude 3.7
Statements (%)	60.85	70.2	72.77	75.24	76.82	83.78
Branch (%)	42.48	56.22	58.75	61.2	63.21	76.27
Functions (%)	72.71	80.47	82.82	83.84	85.7	89.18
Lines (%)	66.62	75.33	77.423	78.97	80.09	86.07
Passing/Failing	165/0	243/40	261/102	302/88	297/48	421/71

In summary, Claude 3.7 Sonnet delivers the most effective amplification results across all evaluation dimensions. Claude 3.5 and Gemini 2.0 Flash follow closely, offering strong performance with fewer test failures. GPT-4o performs reliably but introduces more test-level instability. ChatGPT O3 Mini provides the smallest gains, reflecting its limitations in code generation depth..

5.2.6 Full test amplification run

Based on the results of the prompt and model evaluation, a final full test run is conducted using the best-performing prompt ("Increase Coverage") in combination with three selected LLMs: ChatGPT 4o, ChatGPT O3 Mini, and Claude 3.7 Sonnet. These models are chosen for further analysis based on distinct performance profiles and strategic relevance.

ChatGPT O3 Mini is included due to its lightweight architecture and coding orientation. Despite producing lower coverage scores in the small-scale experiments, it demonstrates a strong pass/fail ratio and consistent behaviour, making it a compelling candidate for minimal-error amplification in constrained environments.

ChatGPT 4o is selected to allow direct comparison with the earlier amplification run conducted using a different prompt, which previously yielded a 6% improvement in branch coverage. This enables evaluation not only of model capability, but also of the relative effect of prompt formulation when using the same model across full-scale amplification.

Claude 3.7 Sonnet is included as the top-performing model from the small-scale experiments. It consistently achieves the highest coverage metrics and the largest volume of valid test cases, making it a strong benchmark for evaluating the upper bounds of LLM-based amplification performance.

The results from these full test runs are compared with each other to determine the most effective LLM configuration. In addition, they are analysed in the context of the search-based test amplification techniques previously introduced, enabling a comprehensive assessment of both symbolic and generative approaches to test suite enhancement for Solidity smart contracts.

5.2.7 LLM-Based Amplification with o4 (Long Prompt)

General Overview

While the motivation and structure of the prompt have already been discussed, this section now presents a full evaluation of the resulting amplified tests.

The first LLM evaluated in this study is OpenAI's o4 model, prompted using a detailed and highly constrained instruction set. This setup mirrors the conditions under which the initial testbench was created, allowing for a fair assessment of how well the model can extend upon its own prior output. The goal of this experiment is to test the LLM's ability to amplify test cases.

The prompt explicitly instructs the model to avoid modifying or removing existing tests, to append only new tests that align with the current style, to follow syntax and compatibility constraints for different Solidity versions, and to favour certain syntaxes when writing tests. By clearly specifying these conventions, the prompt mitigates common pitfalls observed in earlier LLM-generated test sets, such as misuse of outdated function calls or style inconsistencies.

The prompt used is included below:
Here is a Solidity contract 2018- *** .sol and its test file named 2018- ***** -test.js in the hardhat_testing/test directory. Add additional unit tests that will increase the test coverage of the contract. Do not modify or remove any existing tests. Only append new tests that follow the existing test style. Never use .deployed() will cause a TypeError! Only use .deployed() for contracts compiled with Solidity 0.6.x or higher. Also always use ethers.parseEther. Don't ever use '.to.be.revertedWith()' but rather use '.to.be.reverted' for consistency.**

Table 5.4: Coverage improvement with O4 Amplified

Coverage Metric	Baseline (%)	O4 Amplified (%)	Δ (Difference)
Statement Coverage	45.59	51.31	+5.72
Branch Coverage	32.20	39.88	+7.68
Function Coverage	58.50	64.87	+6.37
Line Coverage	52.99	59.12	+6.13

Across the full testbench of 113 contracts, the o4 model generated a total of 685 additional test cases. From these, 368 executed successfully, resulting in a pass rate of 53.72%. While more than half of the test cases pass, the result is not very great. The LLM had a very constrained prompt so common mistakes should have been avoided, and the pass rate should have been higher. Upon manual inspection of the test cases, it was clear that the LLM occasionally ignored the common pitfalls given by the prompt, and just went its own way, which resulted in many failing tests. The test inflation factor for this method is 28.21% indicating a decent increase in test volume relative to the original base.

Coverage Results

Once the newly generated tests are integrated, the suite is re-evaluated using the same four coverage metrics as before: statement, branch, function, and line coverage. The results are presented below in Table 5.4.

Branch coverage increases by +7.68%, followed by function coverage at +6.37%, line coverage at +6.13%, and statement coverage at +5.72%. These figures indicate that the LLM, despite the constraints imposed by the detailed prompt, is still capable of generating structurally meaningful test cases. The improvements are moderate but provide a strong reference point to compare the other LLMs within the following sections.

When analysed in relation to the test inflation factor of 28.21%, the results demonstrate a reasonably efficient amplification process. Although the prompt is designed to minimise syntactic errors and guide test generation towards correctness, the pass rate of 53.72% reveals that this guidance is not always followed. As such, a portion of the generated tests fails to contribute to actual execution success, slightly limiting the effective impact of the increased test volume. This contributes to the low inflation factor. The original number of tests per contract is on average around 9 to 10, so an increase of 28.21% means that each contract now has 2 to 3 additional tests, which is not that significant.

Nevertheless, the structural improvements achieved through this method confirm that even a partially successful integration of prompt constraints can support broader code exploration. These results also highlight the importance of balancing prompt specificity with generative flexibility: while strict prompts can improve test correctness in theory, they may also restrict the LLM's ability to explore novel paths or respond to unexpected code patterns.

5.2.8 LLM-Based Amplification with o4 (Simple Prompt)

General Overview

In this experiment, the same o4 model is reused; however, instead of providing a highly detailed instruction set, it is prompted with a simpler and more open-ended command. The aim of this setup is to investigate how the quality measured by pass rate and coverage metrics of test amplification are affected when the model is given fewer constraints. Preliminary results from the small-scale study (see Section 5.2.3) indicate that this lighter prompt results in fewer errors and improved alignment with the structure of the target contract.

The simplified prompt used in this experiment is as follows:

Table 5.5: Coverage improvement with O4 (Simple Prompt)

Coverage Metric	Baseline (%)	O4 (Simple Prompt) (%)	Δ (Difference)	Δ (Previous Difference)
Statement Coverage	45.59	54.98	+9.39	+5.72
Branch Coverage	32.20	44.43	+12.23	+7.68
Function Coverage	58.50	69.59	+11.09	+6.37
Line Coverage	52.99	62.99	+10.00	+6.13

Here is a Solidity contract 2018-.sol and its test file named 2018–test.js in the hardhat_testing/test directory. Write an extended version of the test class that includes additional unit tests that will increase the test coverage of the class under test.

Unlike the long-form prompt, this version does not include any syntax rules, compatibility warnings, or style guidelines. It simply informs the model about the context, where it can find the contract and test file, and gives it a single objective: increase coverage. The assumption is that by removing rigid constraints, the model gains more freedom to explore relevant behaviours, especially in areas where test opportunities are not immediately obvious from surface patterns alone.

Across the same 113-contract testbench, this setup led the model to generate 1091 amplified test cases. This is significantly higher than the 685 from before, which used the same LLM but a more complex prompt. This observation already highlights one of the strengths of a simpler prompt. Out of these, 519 were valid, resulting in a success rate of 47.58%, a slight decrease from the previous prompt. This confirms that, relatively speaking, overly detailed prompts can still yield more passing tests, while the total number of generated tests is lower.

The test inflation factor here was 36.99%, indicating that the model produced a higher number of new, passing tests to its long-prompt counterpart. Noticeably, contracts that previously yielded poor or no results due to strict syntax constraints benefitted from this freer generation process, where the LLM could just do its own thing.

Coverage Results

As with previous methods, the newly generated tests were evaluated using statement, branch, function, and line coverage. Results are summarised in Table 5.5 above.

Marked improvements are observed across all four metrics. The most significant gains are noted in branch coverage (+12.23%) and function coverage (+11.09%), indicating that the simpler prompt enables the model to explore more diverse conditional branches and more functions. Statement and line coverage also increase substantially, by +9.39% and +10.00%, respectively, reflecting a broader engagement with code structure compared to the baseline.

When viewed alongside the test inflation factor of 36.99%, the efficiency of this method becomes more evident. Although the rate of passing tests is slightly lower than under the complex prompt, the structural gains are significantly higher. This suggests that the simpler prompt leads to a better return on test volume in terms of number of passing tests, and code coverage. While this experiment lacks the strict compliance benefits of the long prompt, the simple approach excels in flexibility and is overall more effective. Tests generated this way often include different approaches to test certain functions, rather than sticking to the exact structure of the original test cases.

Noticeably, the amplified tests rarely contain the mistakes that were explicitly mentioned in the long prompt. This means that, given enough context and a simple prompt, the LLM will refrain from making these mistakes anyway. By analysing the original test cases, the LLM can view that these tests already behave to the wanted assumptions. This allows the LMM to simply align the new test cases with the existing ones, resulting in few to no syntactical errors.

Intermediate Amplification Strategy Evaluation

The comparison between long and simple prompts reveals a clear trade-off. The long prompt enforces consistency but may inadvertently constrain the LLM's generative freedom. Surprisingly, this leads to more syntactic errors and overall a lower pass rate. This is possibly due to the fact that the LLM tries to adhere to the defined syntaxes and rules, rather than taking the style of the already existing tests, and copying this style for the amplified tests. The simple prompt, on the other hand, sacrifices strict alignment in favour of exploration, and this turns out to be beneficial in practice.

In conclusion, the o4 model, when used with a lightweight, coverage-oriented prompt, demonstrates an impressive ability to amplify existing test suites with meaningful new cases. Because of these findings, next sections about the other LLMs will only use the simplified version of the prompt, to give the LLM more generative freedom.

5.2.9 LLM-Based Amplification with o3-mini (Simple Prompt)

General Overview

In this setting, GPT o3-mini is evaluated using the same minimal prompt that proves most effective in the Small-Scale Study. This LLM is significantly more lightweight than o4 and is optimised for speed. The objective of this experiment is twofold: to assess whether smaller LLMs can produce useful test amplifications, and to evaluate the performance-cost trade-off compared to larger models such as o4.

The prompt provided to o3-mini was identical to that used for o4 in the simple prompt setup. This straightforward formulation allows the model to freely generate test cases without being burdened by explicit constraints. Despite having a smaller model size and less contextual awareness than o4, o3-mini still manages to generate meaningful test cases.

In total, o3-mini generated 634 amplified test cases across the 113-contract benchmark. Of these, 388 executed successfully, resulting in a success rate of 61.2%. Surprisingly, this is significantly higher than what was observed with o4 (both prompts), This, paired with the faster generation times, makes o3-mini especially suitable for quick iteration across large datasets.

The test inflation factor was 29.60%, almost identical to the inflation factor of o4 with the constrained prompt. This indicates that o3-mini can be capable of producing a similar volume of test cases. While it doesn't get close to the number of test cases of o4 with the simple prompt, it still shows that o3-mini can effectively generate a good number of test cases.

Coverage Results

The newly generated test suites were again evaluated against the standard coverage metrics. Table 5.6 below shows the results alongside the baseline values and corresponding deltas.

Branch coverage sees the most significant increase at +14.17%, followed closely by function coverage (+13.38%), line coverage (+12.85%), and statement coverage (+12.67%). These improvements are a significant improvement upon the baseline, despite o3-mini generating a relatively lower number of tests. This indicates that, when prompted freely, o3-mini is capable of thoroughly exercising a wide range of code paths.

Table 5.6: Coverage improvement with O3 (Simple Prompt)

Coverage Metric	Baseline (%)	O3 (Simple Prompt) (%)	Δ (Difference)
Statement Coverage	45.59	58.26	+12.67
Branch Coverage	32.20	46.37	+14.17
Function Coverage	58.50	71.88	+13.38
Line Coverage	52.99	65.84	+12.85

The test inflation factor of 29.60% is nearly identical to that of o4 with a constrained prompt, yet the resulting coverage gains are notably higher. This suggests that o3-mini is more efficient in converting test volume into structural impact. Although the model performs slightly less effectively on complex contracts (larger files), the results confirm that compact language models can still make meaningful contributions to automated test amplification. The relatively high pass rate (61.2%) combined with broad coverage gains further supports the suitability of o3-mini for lightweight, prompt-based test amplification, particularly in contexts where speed and efficiency are prioritised.

From a strategic standpoint, o3-mini stands out for its speed. Its ability to quickly produce a large chunk of runnable tests in a short time makes it highly practical for large-scale automation workflows. The simple prompt used here maximised the model's potential, enabling it to focus on coverage rather than conformance.

5.2.10 LLM-Based Amplification with Claude 3.7 (Simple Prompt)

General Overview

Claude 3.7 was the final LLM included in our large-scale evaluation. This model stands out as the slowest among all tested LLMs, with generation times often taking several minutes per contract. Despite its latency, Claude 3.7 proved highly effective in terms of output quality and volume. It consistently produced longer, more elaborate test cases, with deeper reasoning about contract structure and more comprehensive coverage of edge cases.

As with GPT o4 and o3-mini, the same minimal prompt was used for consistency. This simple prompt encourages free-form test generation, which Claude 3.7 handled well. It often introduced logical groupings of test scenarios, under various different 'describe' sections in the javascript code, methodically expanded on parameter spaces, and created well-structured assertions. So far, Claude is the only LLM that took this approach when amplifying test cases.

In total, Claude 3.7 generated 1679 amplified test cases across the 113-contract benchmark. Of these, 1412 executed successfully, yielding a pass rate of 78.14%. This is by far the highest success rate observed across all LLMs, demonstrating not just breadth of generation, but also correctness with the existing Hardhat setup.

The test inflation factor is 60.47%, the highest of all tested models, reflecting Claude's tendency to expand each scenario with multiple subcases and variations. While this increases test volume, it also boosts the probability of uncovering deeper paths through the contract logic.

Coverage Results

Coverage was assessed using the same four standard metrics. The detailed results are listed in Table 5.7 below.

Branch coverage exhibits the most substantial gain at +29.03%, indicating strong effectiveness in exploring alternative execution paths. Function coverage also increases markedly by +20.33%, demonstrating Claude's capacity to trigger a wide range of contract-level behaviours. Similar improvements are observed in statement (+20.31%) and line coverage (+20.07%), further underscoring the depth of the generated tests.

These gains are accompanied by a test inflation factor of 60.47%, the highest among all tested models. This reflects Claude's tendency to generate multiple nuanced variations for each test

Table 5.7: Coverage improvement with Claude 3.7 (Simple Prompt)

Coverage Metric	Baseline (%)	Claude 3.7 (Simple Prompt) (%)	Δ (Difference)
Statement Coverage	45.59	65.90	+20.31
Branch Coverage	32.20	61.23	+29.03
Function Coverage	58.50	78.83	+20.33
Line Coverage	52.99	73.06	+20.07

scenario. While this results in a larger test volume, it also substantially enhances structural exploration, enabling the model to identify corner cases, as well as subtle logic branches that other LLMs may miss.

The exceptional coverage results align with the high pass rate of 78.14%, indicating that the generated tests not only explore more of the codebase but also do so with a high level of correctness. Claude 3.7's ability to group test logic and expand input combinations contributes to a more complete test suite. Although generation times are slower, the output quality and structural coverage make it especially suited for thorough and high-confidence test amplification tasks.

The key trade-off with Claude 3.7 lies between depth and speed. Its outputs were the most verbose among the evaluated LLMs, yet the generation time per contract was significantly higher. Nevertheless, where maximum coverage is prioritised over speed, Claude 3.7 offers excellent results.

5.2.11 Side-by-side comparison

Coverage Metrics

The large-scale evaluation of LLM-based test amplification revealed distinct strengths and limitations among the four tested models: GPT o4 (constrained and simple prompts), o3-mini, and Claude 3.7. All LLMs demonstrated measurable improvements over the baseline across coverage metrics, though their strategies, outputs, and efficiencies varied substantially. These differences can be largely attributed to model size, prompt complexity, and underlying architectural design.

The evaluation of three large language models under the same simple prompt formulation, designed to encourage broad and unconstrained test generation, reveals important trade-offs in terms of test volume, pass rate, and coverage impact. While all three models demonstrate consistent improvements across all four code coverage metrics, their effectiveness varied markedly.

Table 5.9 is a summary between the three LLMs that use the same prompt. It shows all details about the tests, such as the total number of tests, the number of passing/failing tests, the success rate, test inflation rate and all deltas in coverage metrics, when compared to the baseline metrics (see Table 5.8 for the absolute coverage values). These results allow for a direct comparison between models.

Claude 3.7 vs GPT o4 vs o3-mini: A Direct Comparison

Claude 3.7 delivered the most comprehensive amplification, achieving the highest success rate (78.14%), the greatest number of passing tests (1412), and the most substantial coverage deltas across every metric: +20.31% for statement coverage, +29.03% for branch coverage, +20.33% for function coverage, and +20.07% for line coverage. However, these gains came at a significant cost in both latency and test volume: Claude generated 1679 test cases in total, with a test inflation factor of 60.47%, the highest among all models. This aggressive amplification strategy allowed Claude to explore edge cases in depth, often producing nested describe blocks and expanded parameter spaces, which no other model replicated. The expanded parameter space contributes significantly to coverage, especially in complex contracts. Yet, this also sometimes leads to redundancy within the test suites, which may be less desirable in resource-constrained environments. While generation was slower, the outputs aligned well with the Hardhat testing framework and

Table 5.8: Coverage comparison across models and prompting strategies

Coverage Type	Baseline (%)	O4 (Long Prompt) (%)	O4 (Small Prompt) (%)	O3-mini (%)	Claude 3.7 (%)
Statement	45.59	51.31	54.98	58.26	65.90
Branch	32.20	39.88	44.43	46.37	61.23
Function	58.50	64.87	69.59	71.88	78.83
Line	52.99	59.12	62.99	65.84	73.06

Table 5.9: Test and coverage comparison across models for simple prompt

Model	Total Tests	Passing Tests	Failing Tests	Success Rate (%)	Test Inflation (%)	Δ Statement	Δ Branch	Δ Function	Δ Line
Claude 3.7	1679	1412	267	78.14	60.47	+20.31	+29.03	+20.33	+20.07
GPT-4o	1091	519	572	47.58	36.99	+9.39	+12.23	+11.09	+10.00
O3-mini	634	388	246	61.20	29.60	+12.67	+14.17	+13.38	+12.85

demonstrated strong reasoning over complex logic paths, making Claude particularly valuable when depth and structure are prioritised over speed.

In contrast, GPT o4 was evaluated using two prompting strategies. The constrained prompt focused on formatting precision and syntactic safety but led to modest coverage gains due to limited exploration. In comparison, the simple prompt, which allowed greater freedom, resulted in a broader, more exploratory output. Using this configuration, o4 generated 1091 test cases, with 519 executing successfully (47.58% success rate) and a test inflation factor of 36.99%. Although its coverage improvements were solid (+9.39% statement, +12.23% branch, +11.09% function, +10.00% line), it struggled at times with integrating syntactically valid output into the test harness. Nonetheless, its generation speed was faster than Claude's, and its freedom-driven prompting allowed it to explore more diverse logical paths, making it suitable for broad exploration when runtime is a concern.

Finally, o3-mini produced 634 test cases, the fewest among the three, with 388 successful executions, yielding the highest success rate of the trio at 61.20%. Its test inflation factor was 29.60%, indicating smaller yet efficient amplification. Despite its smaller context window and model size, o3-mini delivered surprisingly strong coverage gains: +12.67% statement, +14.17% branch, +13.38% function, and +12.85% line. Its outputs were concise, stylistically consistent with existing tests, and well-suited for contracts with straightforward logic. While it lacked deep contextual adaptation, it made up for it with speed and consistency, making it particularly effective for rapid iteration scenarios or resource-constrained settings.

Relative Efficiency: Balancing Test Quality and Quantity

When assessing these models not just by raw coverage numbers but by efficiency, defined here as coverage gain relative to test inflation and pass rate, some nuanced dynamics emerge:

- Claude 3.7 is the most thorough and effective in absolute terms, especially for achieving high branch and function coverage. However, its cost is high: in computation time, test suite size, and potential for over-generation.
- GPT o4 hits a middle ground, trading off some correctness for breadth. Its higher test volume with lower success rate makes it less efficient than o3-mini, but still competitive depending on use case. It works best when diversity of coverage is more important than test suite compactness.
- o3-mini is the most efficient model. Despite being smaller and producing fewer tests, it delivers high pass rates and strong coverage improvements. This makes it ideal for practical applications where fast feedback loops in test suites are more valuable than exhaustive amplification.

When using the simple prompt configuration, each LLM demonstrates a distinct position in the trade-off space between coverage depth, execution success, and test volume. Claude 3.7 dominates on pure capability and is ideal when coverage is the absolute priority. GPT o4 offers balanced breadth but is less efficient due to a higher failure rate. o3-mini is more limited in generation capacity, but delivers surprisingly high coverage with fewer, more reliable tests. This makes it highly attractive for scalable test amplification.

Table 5.10: Coverage improvement with Random Search

Coverage Metric	Baseline (%)	Random Search (%)	Δ (Difference)
Statement Coverage	45.59	46.00	+0.31
Branch Coverage	32.20	35.78	+3.58
Function Coverage	58.50	58.59	+0.09
Line Coverage	52.99	53.18	+0.19

5.3 Search-Based Amplification

This section will discuss the results for the search-based amplification strategies. The baseline used here is the same baseline that the LLM-based methods used, namely the results from Section 5.1.

5.3.1 Pure Random Search

General overview

The random search amplification method is designed to introduce naive, type-preserving mutations into existing test cases. These mutations are applied without relying on contract semantics, code analysis, or any form of feedback from test execution (static analysis).

Across the full testbench of 113 contracts, the random search procedure generated a total of 6378 new test cases. Of these, only 967 tests compiled and executed successfully, while 5411 tests either failed at runtime or were discarded due to invalid syntax or incompatible input values. This results in a pass rate of 17.87%, indicating that while the approach is broadly applicable, its lack of semantic awareness leads to a large proportion of invalid tests. Before computing the coverage, all failing tests were removed from the testbench to avoid incorrect results.

In addition to basic execution statistics, the amplification process was also evaluated for its test inflation factor. For random search, this factor was 564.12%, highlighting a significant increase in test volume without corresponding structural guidance.

Coverage Results

Following the application of random search, the test suites were re-evaluated using the same coverage metrics as for the baseline: statement, branch, function, and line coverage. Table 5.10 presents the average coverage values obtained after amplification, along with the difference relative to the original testbench.

The results show limited gains from the random search approach. Branch coverage increases modestly, reflecting cases where input mutations resulted in new conditional paths being exercised. However, the remaining metrics show negligible changes; slight increases. Importantly, no original tests were removed during amplification, so only positive differences are possible. Overall, these results highlight the limitations of unguided random amplification. Despite the increased volume of test cases, structural exploration remains shallow, reinforcing the need for more targeted and feedback-aware approaches.

Notably, these marginal coverage changes must be interpreted in light of the test suite's growth. The amplification process increased the number of tests by 564.12%, indicating a substantial rise in test volume. Despite this, the additional tests contribute only minor structural gains. This reinforces the central limitation of random search: its lack of guidance or feedback leads to inefficient test exploration, where many of the generated tests either duplicate existing behaviours or fail to meaningfully expand coverage.

5.3.2 Guided Random Search

General overview

Guided random search builds directly on the principles of pure random search but integrates lightweight heuristics to make the mutation process more targeted. Additionally, input values are selected using a semi-random strategy that biases towards edge cases, while maintaining a degree of randomness. This bias is intended to increase the likelihood of triggering new execution paths, particularly those guarded by conditionals.

Across the 113 contracts in the testbench, guided random search generated a total of 4076 amplified test cases. Of these, 780 compiled and executed successfully, while 3296 were discarded due to runtime errors or syntactic issues, yielding a pass rate of 19.14%. The test inflation factor for this method was 441.60%, reflecting a slightly smaller test volume expansion when compared to the pure random approach, but with more deliberate selection mechanisms.

Coverage Results

After amplification with guided random search, the test suites were re-evaluated using the same four metrics as before: statement, branch, function, and line coverage. The results are shown below in Table 5.11, alongside the baseline values and their respective differences.

The data shows consistent improvements across all metrics. The most notable gains are seen in branch coverage (+7.59%) and line coverage (+5.94%), suggesting that the guided mutations were more effective in exercising conditional paths and broader code segments. Function and statement coverage also benefit, albeit to a slightly lesser degree.

When viewed in relation to the test inflation factor, the guided approach appears significantly more efficient than pure random search. While generating a smaller number of test cases, the structural improvements are more pronounced. This validates the hypothesis that even simple heuristics, such as prioritising control-relevant lines or favouring edge-case mutations, can meaningfully enhance the effectiveness of random test amplification without requiring contract-specific analysis.

5.3.3 Genetic Search

General overview

The genetic search algorithm represents the most advanced test amplification strategy applied in this study. Unlike the random and guided search methods, which apply mutations in a stateless manner, genetic search introduces evolutionary principles: it builds successive generations of test cases through selection, mutation, and crossover. Starting from the original test suite (defined as generation 0), each new generation aims to improve coverage by combining promising test behaviours from earlier iterations.

Due to practical limitations in the Solidity coverage tools, most notably the inability to trace coverage improvements back to individual test cases, the genetic search applies a batching strategy for test evaluation. Each generation's test cases are split into four equal subsets per contract, with each subset evaluated separately. If a batch contributes to coverage, it is retained in its entirety. While this may retain some non-contributing tests, as will be visible in the number

Table 5.11: Coverage improvement with Guided Random Search

Coverage Metric	Baseline (%)	Guided Random Search (%)	Δ (Difference)
Statement Coverage	45.59	52.14	+6.55
Branch Coverage	32.20	39.79	+7.59
Function Coverage	58.50	63.05	+4.55
Line Coverage	52.99	58.93	+5.94

of tests that are generated and retained, the approach significantly reduces runtime and enables scalable analysis.

Metrics in this section are reported per generation. This includes the total number of tests, the pass/fail ratio, the test suite growth rate compared to generation 0, and the average code coverage across four dimensions: statement, branch, function, and line coverage. Since the number of generations is large and coverage changes tend to diminish over time, only selected generations are shown in detail, specifically, generation 1, 2, 5, and 12, to illustrate the trajectory of the amplification process without overwhelming the analysis with exhaustive data. As for generation 5, there is no special reason why it was chosen to display, but it makes sense to not discuss all generations separately. generation 5 was still among the earlier generations, and highlights the path that the amplification process is taking.

Coverage Results: Generation 1

The first generation of genetic amplification, derived from the original test suite through guided mutation and crossover, produced a modest but measurable improvement in overall coverage. Across all 113 contracts, average statement coverage rose to 45.74%, and branch coverage increased to 33.01%. Function coverage remained steady at 58.5%, while line coverage experienced a slight increase to 53.05%. These results are summarised in table 5.12 below, alongside the baseline values and their differences.

The gains, though minor in this first generation, indicate that the mutation and crossover strategy begins to yield new behaviours not present in the baseline suite. These early results validate the potential of genetic search to incrementally improve coverage in a controlled and measurable fashion. Subsequent generations are expected to build on this foundation more substantially.

The relatively modest increase in coverage observed during the first generation can be attributed to structural limitations within the original test suite. Many functions are initially exercised by only a single test case, making it difficult to apply crossover operations effectively. For crossover to produce meaningful new behaviours, a sufficiently diverse pool of valuable tests targeting the same function is required. However, such diversity does not yet exist at this stage. Instead, the primary source of new behaviour must come from random mutations. Only once these mutations manage to expose new paths will test cases be retained during the selection phase and become candidates for crossover in later generations. As such, the small yet consistent coverage gains in Generation 1 are expected and reflect the preparatory nature of this phase in the evolutionary cycle.

In total, 4988 test cases were generated in this generation, of which 3780 passed and 1208 failed, yielding a pass rate of 75.8%, a substantial improvement compared to earlier search strategies, where pass rates remained below 20%. This high success rate highlights the increasing robustness of generated tests as they are built upon a growing foundation of valid patterns. However, these figures precede the selection step. Once failing tests were discarded and coverage-based filtering was applied via the batching strategy described earlier, only 790 test cases were ultimately retained. This means that only 15.84% of initially generated tests are retained (pass rate of 18.82%), leaving behind a smaller, more valuable subset of tests that contributed positively to coverage. It is important to note that not all retained tests are guaranteed to be individually

Table 5.12: Coverage improvement with Generation 1

Metric	**Baseline (%)**	**Generation 1 (%)**	**Δ (Difference)**
Statement Coverage	45.59	45.74	+0.15
Branch Coverage	32.20	33.01	+0.81
Function Coverage	58.50	58.50	±0.00
Line Coverage	52.99	53.05	+0.06

beneficial, but were preserved as part of larger batches that included at least one coverage-improving case (see Section 4.6.7 about batching technique for selection in genetic search).

Compared to the random and guided search strategies, the genetic search exhibits a markedly more restrained test suite expansion in its first generation. Starting from 923 original tests, only 790 new tests are retained after filtering out failing cases and applying selection based on coverage contribution. This corresponds to a test suite of 1713 tests, with a growth of 85.6%, which is significantly lower than the growth rates of 564.12% and 487.75% observed for random and guided search respectively. This difference reflects the deliberate design of the genetic search algorithm, which prioritises the retention of high-value test cases over sheer volume. By applying coverage-based selection and discarding uninformative or redundant tests, the approach ensures that each additional test contributes meaningfully to behavioural exploration. As such, genetic search offers a more targeted path to amplification, trading off quantity for precision.

Coverage Results: Generation 2

The second generation of the genetic search algorithm yields a modest yet consistent improvement over the baseline in terms of coverage. As shown in Table 5.13, all four metrics show incremental gains: statement coverage rises from 45.59% to 45.78%, branch coverage from 32.20% to 33.63%, function coverage remains constant at 58.5%, and line coverage increases slightly from 52.99% to 53.04%. These small deltas reinforce the iterative nature of the genetic approach, where gradual refinements are accumulated across generations

From a test volume perspective, the second generation introduces a much larger number of test cases. Before selection, 12668 tests are generated, of which 10397 pass and 2271 fail, resulting in a pass rate of approximately 82.1%. After removing failed tests and applying the selection mechanism, only 1939 tests were retained. This selective retention reflects the algorithm's focus on identifying valuable contributions to coverage rather than meaninglessly expanding the test suite. Despite the large number of initial candidates, only 15.31% of the generated tests were kept (pass rate of 18.60%).

The second generation of genetic search continues to expand the test suite substantially. Starting from the baseline of 923 original tests, generation 1 added 790 selected tests, bringing the total to 1713. With generation 2, another 1939 tests were retained after the selection phase, resulting in a new total of 3652 tests. This corresponds to a test suite expansion of approximately 295.7% relative to the original suite. Such growth reflects the increasing volume of tests being generated and filtered through the genetic algorithm's selection process. While only a fraction of generated tests are ultimately kept, 1939 out of 10397 passing tests in this case, the cumulative effect across generations is a steadily growing suite that prioritises meaningful coverage gains. This gradual build-up of valuable test cases lays the groundwork for deeper exploration of contract behaviour in future generations. A small remark here is that tests from different generations can be combined, but are maintained in different test files and folders to separate generations from each other.

Coverage Results: Generation 5

Across all four metrics, generation 5 shows consistent improvements over the baseline, as shown in Table 5.14. The most substantial gain appears in branch coverage (+7.62%), which aligns with

Table 5.13: Coverage improvement with Generation 2

Metric	Baseline (%)	Generation 2 (%)	Δ (Difference)
Statement	45.59	45.78	+0.19
Branch	32.20	33.63	+1.43
Function	58.50	58.50	+0.00
Line	52.99	53.04	+0.05

Table 5.14: Coverage improvement with Generation 5

Metric	Baseline (%)	Generation 5 (%)	Δ (Difference)
Statement	45.59	51.87	+6.28
Branch	32.20	39.82	+7.62
Function	58.50	62.93	+4.43
Line	52.99	58.80	+5.81

the intent of the genetic algorithm to explore complex control flows via iterative selection and crossover. Similarly, statement and line coverage improve by over 6% and nearly 6% respectively. Function coverage also rises moderately, suggesting that more distinct contract functions are being exercised, though perhaps not with deeply varied inputs. This is not directly possible as the implementation of the genetic search doesn't parse the Solidity contracts, but rather by exercising new branches that call certain functions which weren't tested in previous generations. These cumulative improvements reflect the algorithm's increasing ability to build upon valuable prior tests.

In generation 5, the genetic search algorithm reaches a phase where test generation becomes significantly more prolific, yet increasingly selective. The increasingly selective nature has been observed in several other studies that employ genetic algorithms, among them the study by Akca, S., et al. [46]. In this study, the number of retained tests keeps decreasing as time goes on. Intuitively, this also makes sense because the more valuable tests that are being generated, the harder it becomes to find new tests that contribute positively to the coverage metrics. This makes generation 5 an interesting observation point in the genetic process.

A total of 40002 tests were generated, comprising 35128 passing and 4874 failing tests. From these, only 11247 tests were retained after selection, which is 28.12%. This reflects the algorithm's growing emphasis on preserving only those batches that contribute new or unique coverage. As noted earlier, the batching approach groups tests into sizable units, now often nearing 100 tests per batch in some cases. Although most of these batches are discarded due to redundancy, a few are still retained entirely when at least one test within them proves valuable. This results in some duplication, as non-contributing tests within a retained batch remain in the suite. Nonetheless, this trade-off is accepted in favour of practical feasibility.

The retained 11247 tests, combined with the 3652 tests from generation 0, 1 and 2, and the 4077 and 6173 tests from generation 3 and 4 respectively, which weren't discussed but are visible in Table 5.15, the total test suite size 25149. This is an average of 223 tests per contract. While executing these tests in Hardhat is not a problem, in terms of execution time, the redundant tests in the dataset can influence the effectiveness of genetic search in future generations. Additionally, the significant increase in volume contrasts with the relatively modest improvements in coverage observed at this stage, suggesting diminishing returns in raw metrics.

Table 5.15: Genetic search results per generation

Generation	Generated	Passed (%)	Retained	Retention (%)
1	4988	3780 (75.8%)	790	15.84
2	12668	10397 (82.1%)	1939	15.31
3	19903	16599 (83.3%)	4077	20.48
4	26570	22850 (86.0%)	6173	23.23
5	40002	35128 (87.8%)	11247	28.12

Coverage Results: Generation 12

This is the comparison between the baseline and generation 12, which is the final generation for the genetic search algorithm. The decision to stop at generation 12, is based on the stopping criteria, stated in Section 4.6.10. The increase in branch coverage between generation 5 and

12 is only 2.81%, which is quite insignificant when compared to the branch coverage increase between generation 2 and 5, which was 6.19%. The concrete stopping criterium was that when the branch coverage increase between two generations is less than 0.25%, the evaluation stops. The difference in branch coverage between generation 11 and 12 was 0.24%, falling just below the required 0.25% to proceed to the next generation.

However, the final generation shows substantial improvements across all metrics compared to the baseline, as shown in Table 5.16. The most notable gain is in branch coverage, which improved by 10.43%, underscoring the effectiveness of the genetic search in navigating complex control flow paths. Statement and line coverage also increased by 7.18% and 6.51%, respectively, indicating that the amplified tests execute broader portions of the codebase. Function coverage sees a moderate yet consistent rise of 4.55%, suggesting that the test suite explores a more diverse set of functions.

After 12 generations, the testbench expanded to a total of 128816 passing tests, a stark increase from the 923 tests in the original suite. This dramatic growth is largely driven by the batching technique used during selection. While batching allows for efficient coverage evaluation without isolating the impact of each individual test, it also introduces a high degree of test duplication. As many batches contain tests with overlapping behaviour, and crossover operations are often applied to similar or near-identical parents, the resulting test pool becomes increasingly saturated with redundant inputs. Ideally, selection would use a per-test fitness function to retain only the most valuable additions, but batching precludes this level of granularity. The result is an exponential blow-up in test volume, particularly noticeable from generation 9 to 12, where the number of tests nearly doubled with only marginal gains in coverage.

However, these final results confirm that the genetic algorithm is capable of systematically improving structural coverage through guided test generation and iterative selection. As the improvements between later generations began to diminish, generation 12 was chosen as the practical endpoint. If a per-test fitness function approach was available, the number of generations would likely be pushed further than 12 but it's unclear when convergence would occur.

5.3.4 Side-by-side comparison

Coverage Metrics

To conclude the coverage analysis of the different amplification strategies, it is useful to present a side-by-side comparison of their performance across all four coverage metrics. This allows us to clearly observe how each method contributes to increasing test coverage relative to the original test suite, and to identify the added value of incorporating more sophisticated techniques such as heuristics and evolutionary algorithms.

As shown in the Table 5.17, pure random search yields only a marginal improvement over the baseline, indicating that uninformed mutations have limited effect on exercising new paths. The guided random approach, by introducing heuristics to better target control-flow-relevant inputs, achieves a significantly higher coverage increase in all coverage metrics. The genetic search, measured at generation 12, slightly improves upon the guided method, with the most notable gain in branch coverage, rising from 35.20% to 42.63%, confirming that selection and crossover help uncover deeper or harder-to-reach paths. Overall, the results confirm that increasing algorithmic sophistication leads to consistent improvements in test coverage.

Table 5.16: Coverage improvement with Generation 12

Metric	Baseline (%)	Generation 12 (%)	Δ (Difference)
Statement	45.59	52.77	+7.18
Branch	32.20	42.63	+10.43
Function	58.50	63.05	+4.55
Line	52.99	59.50	+6.51

Table 5.17: Coverage comparison across search-based methods

Coverage Type	Baseline (%)	Pure Random (%)	Guided Random (%)	Genetic Search (G12) (%)
Statement	45.59	46.00	52.14	52.77
Branch	32.20	35.78	39.79	42.63
Function	58.50	58.59	63.05	63.05
Function	52.99	53.18	58.93	59.50

Guided random search introduces significant improvements over pure random. By applying heuristics that better focus mutation effort on code structures likely to affect coverage, such as targeting functions with many, it manages to boost coverage metrics significantly. Branch coverage sees a particularly notable improvement (+7.59%), indicating that heuristic guidance improves the probability of triggering diverse conditional branches. The gains here validate the usefulness of lightweight guidance mechanisms even in otherwise stochastic systems.

The genetic search algorithm, evaluated after 12 generations, shows decent improvements over the guided approach. Its main advantage lies in branch coverage, where it reaches 42.63%, a 2.84-point increase over guided random and a 10.43-point improvement from the baseline. This suggests that evolutionary operators like selection and crossover can progressively refine inputs to reach harder-to-exercise logic, including complex decision paths and edge-case conditions.

Interestingly, function coverage does not increase beyond the level already achieved by guided random testing (63.05%), suggesting that by generation 12, the search process may have saturated the reachable functional space. Based on the nature of the search-based methods, untested functions cannot be directly targeted unless they are invoked by already-tested functions. As a result, any increase in function coverage depends on the indirect discovery of such calls within the bodies of previously tested functions.

This implies that either all feasible paths leading to untested functions have already been explored by both guided and genetic search, or both strategies coincidentally explored the same set of paths, despite additional, unexplored paths still existing. The former scenario appears more likely. Considering the large number of available functions, it would be statistically improbable for two independent algorithms to reach the same function coverage if additional reachable functions were still present.

However, it remains possible that a small number of functions are reachable only under very specific input conditions that neither technique has yet triggered. In the context of Solidity contracts, this is a plausible explanation. Solidity contracts typically maintain little internal state; many functions, such as ‘mint’, ‘burn’, ‘sell’, ‘buy’, and ‘transfer’, modify state directly without relying on nested internal calls. Consequently, there are relatively few branching points that lead to additional function invocations. This structural simplicity may inherently limit the potential for further function discovery through test amplification.

Therefore further evolutionary pressure mostly benefits branch depth rather than breadth. The modest additional increases in statement (+0.63%) and line coverage (+0.57%) reinforce this idea: while evolutionary refinement helps, its main contribution is in depth rather than volume.

These results confirm a clear pattern: as the sophistication of the search method increases, from pure random to guided to genetic search, so does the effectiveness of the amplification in expanding test coverage. However, the returns are not uniform. Heuristics provide the most immediate improvement, especially in metrics sensitive to control flow like branch and function coverage. Genetic algorithms then add incremental, targeted gains by fine-tuning test cases that are already partially effective.

While the improvements may seem incremental, they are valuable for real-world testing scenarios where branch and function coverage may directly correlate with fault detection capability. Moreover, these methods offer reproducible augmentation pipelines, which is a strength compared to LLM-based techniques that can be more variable and context-sensitive. Nonetheless, the best performance may arise from hybrid strategies that combine the systematic search of these methods with the semantic reasoning of LLMs.

5.4 Test inflation analysis

This section presents a comparative analysis of test suite growth across different amplification techniques, with a primary focus on the test inflation factor the relative increase in test suite size after amplification. While structural coverage improvements (e.g., statement, branch, function, and line coverage) remain an important performance indicator, test inflation provides critical insight into the efficiency and overhead of each method. In particular, it reveals how much additional test code is generated to achieve those coverage gains, helping distinguish between brute-force expansion and targeted amplification.

Although LLM-based methods often match or surpass traditional techniques in coverage especially in function and line coverage the central question here is how economically that coverage is achieved, measured by the volume of test cases produced relative to the original suite. Among search-based methods, genetic search (G12) consistently outperforms Pure Random and Guided Random, especially in branch and function coverage. However, LLM-based techniques, such as Claude 3.7, GPT-4o (simple prompt), and GPT o3-mini, achieve even better results across most metrics. Claude 3.7 reaches 78.83% function and 73.06% line coverage, significantly exceeding Genetic Search (63.05%, 59.50% respectively). GPT o3-mini and GPT-4o also outperform search-based methods in all metrics except branch coverage, where Genetic Search retains a slight edge over GPT-4o.

To evaluate cost-effectiveness, the test inflation factor quantifies the increase in test suite size. pure random and guided random produce large inflations (564.12% and 441.60%), often with redundant tests. Genetic search at generation 2 inflates by 295.7%, and continues growing significantly due to limited filtering, reaching a test inflation factor of 13956.23%, which is incredibly high. So while genetic search is quite effective at increasing coverage, particularly branch coverage, it comes at a high computational cost when doing a batch selection operation. In contrast, LLMs generate more compact and focused expansions. GPT-4o (constrained) results in 28.21% inflation, GPT-4o (simple) in 36.99%, GPT o3-mini in 29.60%, and Claude 3.7 in 60.47% the highest among LLMs, reflecting a more aggressive strategy.

These results show that LLMs offer superior coverage with lower overhead, producing fewer but more effective tests and achieving a better coverage-to-volume ratio than traditional search-based methods.

5.5 The Problem of Fairness in Comparing Search-Based and LLM-Based Test Amplification

A critical challenge arises when attempting to compare search-based test amplification techniques with LLM-based test generation. The core of the issue lies in the disparity of capabilities between both approaches, particularly regarding the way they interact with function coverage in the test dataset. While both aim to improve code coverage metrics such as statement, branch, line, and function coverage, their fundamental operations differ in such a way that may result in an unfair advantage for LLM-based techniques.

Search-based algorithms perform test amplification by causing mutations in existing test cases and recombining them. While they do have access to the source code, they don't do control flow or data flow analysis with it. They rely solely on feedback from metrics, such as code coverage, to guide the search process. In contrast, LLM operate in a slightly different manner. Just like search-based algorithms, they also have access to the source code and they can examine both the Solidity code and its corresponding test suite, in order to create a context. This enables them to generate entirely new test cases targeting specific, previously untested functions, regardless of whether those functions are present in the original test suite.

This difference leads to a significant issue in comparative analysis. For example, consider a scenario in which the original test suite achieves 60% function coverage and 50% branch coverage. Suppose that the functions currently covered by tests contain 70% of the total branches, while

the remaining 30% of branches are located within untested functions. A search-based algorithm, which cannot explore functions absent from the test suite, would theoretically be limited to a maximum branch coverage of 70%, assuming it finds every possible path within the functions it already knows. The remaining branches, tied to untested functions, are inaccessible under the current search model.

By contrast, an LLM-based approach can easily generate a single minimal test per untested function, thereby substantially increasing function and, indirectly, branch coverage. Even if these generated tests are shallow and do not explore complex edge cases, they can significantly affect overall metrics. In practice, this leads to scenarios where LLMs achieve large coverage gains with small test output, while search-based methods produce many more tests that explore deeper logic but yield smaller apparent improvements.

One refinement to this issue is that search-based methods can, in rare cases, indirectly increase function coverage. This occurs when untested functions are invoked along alternative paths within already-tested entry points. This increase, however, is contingent on such a branch being present and discoverable via input mutation. In most cases, untested functions exist outside the reach of such control-flow paths, particularly when they are never referenced in any way within the original test suite.

Furthermore, the constraint of remaining within the boundaries of amplification, rather than entering the realm of test generation, places search-based methods at a disadvantage. Incorporating functionality to parse the Solidity code to identify untested functions would constitute a form of test generation, which violates the original goal of amplifying existing tests. While such parsing is technically feasible, it introduces extra complexity, particularly when dealing with stateful contracts, inheritance, or dependencies on internal state or external libraries. On top of that, the Solidity version of the contracts under test ranges from Solidity v0.4.2 to Solidity v0.6.4, causing the need to write a parser that can handle all of the supported Solidity versions from the dataset. All of this risks shifting away from its core amplification focus.

In conclusion, direct comparison of LLM-based and search-based approaches using raw coverage gains must be approached with caution. The LLM's access to the source code and its ability to create new tests for previously untested functions grants it a systemic advantage. While this reflects a real-world capability of modern tools, it complicates attempts to measure the relative effectiveness of different test amplification strategies in a fair and balanced manner.

5.5.1 Further Evaluation of Results with a more Fair Testbench

The first part of the results section presents a comprehensive evaluation of test amplification techniques using the original testbench. This includes a small-scale investigation of prompting strategies for LLMs, followed by a large-scale assessment of three LLMs and three search-based techniques. These experiments provide insights into the effectiveness, practical limitations, and behavioural tendencies of each approach, as well as their interaction with Solidity smart contracts of varying complexity.

Despite inherent limitations in the initial testbench, such as the structural advantage granted to LLMs through access to untested functions, reliable intra-category comparisons remain possible. All LLM-based methods benefit equally from this advantage, which still enables consistent evaluation of their relative performance. Similarly, all search-based techniques operate under identical constraints, relying exclusively on the existing test cases. These shared conditions allow for the identification of the most effective technique within each category.

For this reason, the second part of the evaluation does not re-examine all amplification techniques. Instead, only the best-performing search-based method and the best-performing LLM-based method are selected for further experimentation. This approach avoids redundancy and focuses the analysis on the primary objective of the study: a comparison between search-based and LLM-based amplification in the context of Solidity smart contract testing.

The augmented testbench, constructed to maximise function coverage, offers a more equitable testing environment. Since most testable functions are now already covered in the baseline, the structural advantage previously observed in LLMs is significantly reduced. As a result, any further improvements in code coverage are more likely to reflect meaningful behavioural testing rather than superficial increases from newly reached functions.

Alongside individual evaluations of the two selected techniques, two hybrid configurations are introduced. One applies search-based amplification first, followed by LLM-based generation; the other applies these steps in reverse order. These hybrid approaches aim to determine whether a combination of paradigms leads to further improvements, leveraging both the targeted input space exploration of search-based methods and the generative flexibility of LLMs. Focusing the second amplification stage on the most effective representatives of each category allows for a better evaluation on the improved dataset.

The next few sections will therefore only be focussed on four more experiments: test amplification using genetic search, using Claude 3.7, and the two hybrid scenarios. All of these experiments will be compared to the new baseline metrics, obtained from the second, more fair, dataset.

5.6 New baseline metrics

Before applying any amplification techniques, the newly augmented baseline testbench is evaluated. This testbench consists of 101 Solidity contracts, drawn from the same diverse dataset as the original benchmark. The selection is filtered to include only those contracts for which meaningful improvement in function coverage can be achieved through augmentation. Each contract is paired with an extended test suite generated using Gemini 2.0, with the goal of reaching function coverage as close to full as realistically possible.

To assess the quality of this new baseline, four standard coverage metrics are measured: statement coverage, branch coverage, function coverage, and line coverage. These metrics provide a comprehensive view of test suite effectiveness, capturing both general code execution and more nuanced behavioural depth. The results are presented in Table 5.18.

These improved coverage values indicate that the augmented baseline achieves a significantly higher level of functional exploration compared to the initial test suite. In particular, the function coverage of nearly 77% reflects the deliberate design goal of covering as many testable functions as possible prior to amplification. This helps reduce the structural bias that favours LLMs in reaching untested functions, and instead shifts the focus towards deeper behavioural testing.

The relatively high branch and statement coverage metrics further support the effectiveness of the augmented tests in traversing conditional logic and code paths. Although not exhaustive, these values suggest that the LLM-generated augmentation introduces meaningful variability in test scenarios.

5.7 Fair Amplification Results

The ordering of the following sections is based on the structure of the experiments rather than a strict separation between standalone and hybrid methods. While it may seem natural to present

Table 5.18: Average coverage metrics

Coverage Metric	**Average (%)**
Statement Coverage	62.26
Branch Coverage	50.49
Function Coverage	76.99
Line Coverage	69.27

all individual techniques first, followed by the hybrid configurations, such an approach would break the logical flow of the testbench construction process.

Each hybrid configuration in this study builds incrementally upon the output of a previous technique. The test suite produced by the first method becomes the input for the second, allowing the second stage to explore and enhance what the first may not have reached. For instance, in the 'first genetic, then Claude' setup, Claude 3.7 receives the output of the genetic algorithm and adds further amplification. The same principle applies in reverse for the 'first Claude, then genetic' configuration.

This cumulative relationship means that the performance of each hybrid reflects the combined effect of both methods. Coverage results from hybrid configurations are therefore always equal to or greater than those of the individual methods they extend. These results are not isolated but represent a layered improvement.

Presenting each standalone method directly before the hybrid that builds on it enables a more straightforward comparison. It allows the reader to observe how coverage improves step by step, and avoids interruptions in the narrative by jumping between unrelated results. This structure also highlights the complementary strengths of both approaches, showing clearly how one method enhances the output of the other.

5.7.1 Re-evaluation of Genetic Search

General Overview

Similar to the workflow in the first testbench, the process begins with the original test suite, which acts as the initial population. Over seven generations, the algorithm produces increasingly diverse test cases by learning from intermediate results. This feedback loop significantly improves the likelihood of generating high-quality inputs that cover difficult-to-reach paths. Importantly, this method benefits most when the initial population includes structurally diverse and semantically relevant test cases, enabling meaningful crossover combinations from the outset.

In the context of the 101 contracts evaluated, genetic search generated a total of 63696 new test cases by the final generation. Out of these, 56182 executed successfully, while 7514 were rejected due to runtime issues, resulting in a pass rate of 88.2%. Given the original testbench contained 1566 tests, this corresponds to a test inflation factor of 4068.5%, indicating a substantial expansion in test suite size. Despite this volume, the high pass rate highlights the algorithm's ability to generate syntactically valid inputs, thanks to its selective reproduction. The reason for this enormous number of tests is again due to the use of batch selection, where sometimes, tens to hundreds of tests can be retained at once because just one test increases one of the coverage metrics.

Coverage results

The genetic search-based technique was again executed over multiple generations, following the same way of working established during the earlier phase of the study. This consistency ensures that the evaluation remains aligned across both datasets. While this study will not do any comparisons directly between the two datasets, it makes sense to follow the same workflow as before to allow for valid comparisons with the LLMs later on. However, the generational approach also provides an opportunity to confirm whether similar patterns of coverage evolution, such as early-stage stagnation and mid-stage acceleration like before, persist under different initial conditions. The coverage results of the testbench for several generations can be found in Table 5.19.

Notably, the augmented testbench revealed an earlier stop of the genetic algorithm. It stops at generation 7, while it stopped at generation 12 during the test amplification for the first dataset. This is due to the other stopping criterium discussed in Section 4.6.10. If the manual labour of the evaluation + selection process exceeds two hours, the algorithm is brought to a

Table 5.19: Coverage progression across generations (Gen 0–7)

Coverage Metric	Base	Gen 1	Gen 2	Gen 3	Gen 4	Gen 5	Gen 6	Gen 7
Statement	61.09%	62.15%	62.93%	63.14%	63.40%	63.67%	64.41%	65.23%
Branch	50.51%	51.70%	54.99%	55.16%	55.69%	55.92%	56.97%	57.44%
Function	74.85%	75.06%	75.18%	76.09%	76.09%	76.37%	77.65%	77.90%
Line	67.85%	68.33%	69.38%	69.89%	69.89%	70.30%	70.99%	71.75%

stop. The exact reason for the longer evaluation process is unclear, but might be due to the fact that Gemini 2.0 occasionally makes the introduction of nested loops in the LLM-generated test cases. These constructs, when subjected to mutation, occasionally expanded loop ranges dramatically, resulting in extended test execution times, which in some cases greatly increased the selection time because tests took a lot longer to run. So while there is a monotonic increase in coverage metrics at this stage of the genetic search, the process was brought to an early stop due to practical limitations.

An important observation from the second testbench is the marked difference in how crossover contributes to coverage improvements across generations, especially when contrasted with the dynamics seen in the first testbench. In the original dataset, early generations were dominated by modest, mutation-driven gains. This was primarily due to the structural sparsity of the initial test suite, where most functions were tested by only a single case, making it difficult for crossover to generate useful recombinations. Without diverse parent test cases targeting the same function, the evolutionary algorithm had limited material to work with, and crossover was largely ineffective until later generations, after mutations had begun to populate the test pool with more variety.

By contrast, in the second testbench, augmented by Gemini 2.0, the situation is reversed. Because the test suite already contains a richer and more structurally diverse set of test cases, including edge-case scenarios for many previously tested functions, the conditions necessary for effective crossover are present much earlier in the evolutionary process. This change is clearly visible when comparing the improvements in branch coverage across early generations. From generation 0 to generation 1, branch coverage improves modestly from 50.51% to 51.70%. However, the increase from generation 1 to generation 2 is much more pronounced, rising sharply to 54.99%. This suggests that crossover is already yielding compound behaviours by this point, made possible by a test pool rich enough to support meaningful recombination.

This pattern contrasts with the relatively flatter coverage deltas observed in later generations, for example, the increase in branch coverage from generation 3 to 4 is only 0.53%, from 55.16% to 55.69%. These diminishing returns in later generations further support the hypothesis that the earlier availability of crossover-ready material accelerates convergence. Thus, the structure of the initial dataset, particularly the diversity and completeness of the starting test pool, has a decisive impact on when and how crossover begins to meaningfully contribute to coverage gains. In the augmented setting, it enables the genetic algorithm to bypass the typically slow, mutation-driven setup phase and begin benefiting from evolutionary mechanisms right from the outset.

Additionally, while the notion of early convergence in the second testbench may appear normal at a glance, a more nuanced examination, particularly of the relationship between function and branch coverage, reveals a deeper insight into the effectiveness of crossover in the early evolutionary stages. One would normally expect that increases in function coverage, causing many new conditional paths to be explored at once, should correlate with big rises in branch coverage. This relationship generally holds true, and it is particularly visible in later generations of the second testbench. However, the early stages display an interesting deviation from this pattern.

Between generation 0 and generation 2, branch coverage increases substantially, from 50.51% to 54.99%, even though function coverage only grows modestly, from 74.85% to 75.18%. This asymmetry strongly highlights the role of effective crossover in exposing deeper conditional logic

within already-covered functions rather than discovering entirely new functions. Thanks to the structurally richer initial test suite, the genetic algorithm is able to recombine diverse parent test cases in ways that explore alternate execution paths and edge cases without necessarily adding new functions.

In contrast, later generations show the opposite pattern. Function coverage starts to rise more noticeably, for example, from 76.37% in generation 5 to 77.90% by generation 7, while branch coverage gains diminish, increasing only from 55.92% to 57.44% over the same period. This suggests that the evolutionary process is shifting its focus from exploring within known functions (through crossover) to discovering new ones via mutations or deeper conditional entry points. However, since many of these newly discovered functions may not contain complex branching logic, or may be simpler utility functions (often private functions that aren't tested in the initial testbench), their contribution to branch coverage is smaller. This again reinforces the observation that early, substantial gains in branch coverage were driven not by the discovery of new functions, but by effective recombination of test inputs that activate previously unexplored branches in existing functions.

Taken together, these findings confirm that the strength of the genetic algorithm in this testbench lies not merely in raw amplification potential but in its early exploitation of a high-quality starting population. This enables it to achieve quick and efficient branch coverage improvements through crossover, before transitioning to slower, mutation-driven exploration of new functions in later generations. Such dynamics highlight the critical role that initial test suite structure plays in shaping the trajectory of genetic amplification techniques.

5.7.2 Hybrid configuration 1: first genetic, then claude

post-processing of the testset

Genetic search is highly effective at generating valuable test cases, but its raw output often suffers from significant redundancy. Across the 101 contracts, the algorithm produced 63696 new test cases, an average of over 600 tests per contract. While high in volume, many of these tests offer minimal additional value, often representing small variations of earlier generations that no longer improve coverage metrics. This high number of tests is due to the use of doing the selection process in batches.

Because of token limitations and practical constraints of large test files, a post-processing phase was required to downscale this volume. This was done via 'redundancy-based batching'. Starting from the final generation, test cases were grouped into small random batches. Each batch was temporarily excluded from the test file, and the resulting coverage was re-evaluated. If coverage remained unchanged, the excluded batch was deemed redundant and permanently removed.

The goal was to bring each contract's test file down to roughly 150 tests, targeting an approximate 75% reduction. Initial batch removals were fast and effective; when most tests are redundant, randomly selecting even for example 30 tests (out of more than 600) is likely to yield a non-contributing subset. However, as coverage-critical tests became proportionally more prevalent, the filtering process slowed significantly.

After extensive iterations, this approach reduced the total test suite to 15958 tests, achieving the target of removing over 75% of tests. The resulting testbench has 158 tests per contract on average. This curated subset then served as the input to Claude 3.7 in the hybrid setup, ensuring a manageable test file size while retaining the maximum coverage from before.

General overview

In hybrid configuration 1, the Claude 3.7 model builds on a refined, post-processed output from the genetic search phase. Starting from a reduced test suite of 15,958 tests, obtained through

redundancy-aware batching, Claude was tasked with further amplifying the suite to explore missed paths and edge cases. The prompt used is the same, small one that was used before.

Claude generated an additional 725 test cases, of which 359 passed and 366 failed, resulting in a pass rate of 49.52%. This rate is notably lower than in previous setups, which can be attributed to the inherently more complex and saturated test input it received. Unlike in the default or LLM-only scenarios, many obvious or easily testable behaviours were already covered by the genetic phase. As a result, Claude had to target deeper or more intricate execution paths, leading to higher failure rates but still producing valuable test cases.

The hybrid approach thus leverages genetic search for broad structural exploration, followed by Claude's semantic understanding to fill in remaining gaps. Despite the low success ratio, the pass cases from Claude were often highly targeted and nuanced, complementing the largely 'black-box' nature of genetic testing with precise, function-aware test logic.

Coverage results

Coverage outcomes for the first hybrid configuration, where genetic search is applied first, followed by Claude, are summarised in Table 5.20, alongside the individual results of the final generation of genetic search alone for comparison. This side-by-side view helps isolate the contribution of the LLM when applied to an already high-performing, search-based-generated test suite.

The results confirm that Claude 3.7 is effective at amplifying an already strong test suite produced by genetic search. Despite starting from a point where much of the low-hanging coverage was already achieved, Claude contributes significant additional gains across all four metrics.

The largest relative improvement is observed in branch coverage (+6.61%) and in function coverage (+6.36%), demonstrating Claude's capacity to test previously untested functions, and generating tests for edge cases that were previously missed. The improvements in statement (+4.58%) and line coverage (+4.48%) further reinforce Claude's ability to increase structural depth, even when operating on an input set refined to reduce redundancy and maximise existing coverage.

In conclusion, while genetic search excels at broad exploration, Claude 3.7 adds depth, nuance, and semantic precision, making this hybrid configuration a compelling solution for high-fidelity test suite amplification. The results also illustrate that effective collaboration between both methods can yield superior results, especially in domains with complex execution paths and subtle control logic like smart contracts.

5.7.3 Re-evaluation of Claude

General overview

Moving on to the LLM-based approach. This method benefits from the model's contextual awareness and pattern generalisation capabilities. Claude 3.7 often proposes semantically rich tests that mimic human-like reasoning, especially in generating edge-case inputs or invoking less frequently tested branches. However, since the generation process does not involve actual execution feedback (or the use of mutations), it is inherently limited to the model's internal knowledge and inference abilities at generation time.

Applied to the same dataset of 101 contracts, Claude 3.7 produced a total of 1671 new test cases. Out of these, 1249 executed successfully, while 422 failed due to syntax errors or invalid

Table 5.20: Coverage improvement after post-genetic evaluation with Claude 3.7

Coverage Metric	Genetic Search Gen 7 (%)	Claude 3.7 (Post-Genetic) (%)	Δ (Improvement)
Statement Coverage	65.23	69.81	+4.58
Branch Coverage	57.44	64.05	+6.61
Function Coverage	77.90	84.26	+6.36
Line Coverage	71.75	76.23	+4.48

Table 5.21: Coverage improvement with Claude 3.7

Coverage Metric	Baseline (%)	Claude 3.7 (%)	Δ (Difference)
Statement Coverage	62.26	71.12	+8.86
Branch Coverage	50.49	67.19	+16.70
Function Coverage	76.99	84.77	+7.78
Line Coverage	69.27	77.38	+8.11

logic, resulting in a pass rate of 74.75%. Relative to the original testbench of 1566 test cases, this corresponds to a test inflation factor of 106.7%, indicating a moderate increase in test suite size. The lower pass rate compared to genetic search reflects the challenges LLMs face in guaranteeing both syntactic and semantic correctness, but the method remains valuable for injecting structural diversity into the test suite without iterative search.

Coverage results

Coverage outcomes for Claude 3.7 on the second, more balanced dataset further reinforce the model's strong amplification capabilities, while also highlighting how different initial conditions can influence overall gains. As shown in the context of a fairer testbench, the improvements remain substantial, though slightly less dramatic than in the previous experiment, due to the more comprehensive baseline. Results are visible in Table 5.21.

Branch coverage again registers the most pronounced gain (+16.70%), showcasing Claude's ability of diversifying conditional logic to reach more control paths. This confirms previous findings, but with greater nuance: the baseline already contains significant structural diversity due to the dataset's improved composition. As a result, while the relative improvement is less dramatic than in the first testbench, the absolute coverage values reached are still notably high.

Statement, line, and function coverage each improve by around 8 percentage points. These consistent gains indicate that Claude 3.7 retains its strength in expanding the behavioural space across different execution levels, from low-level line and statement tracing to high-level function invocation. Function coverage exhibits the smallest relative change (+7.78%), suggesting that many function-level behaviours were already partially exercised in the augmented baseline, reducing room for expansion. This observation underscores the value of our new testbench.

In summary, Claude 3.7 remains a great performer in LLM-based test amplification, even under more balanced conditions. This is obvious by the good increases in coverage metrics. Its ability to produce high-quality test cases that significantly boost both control-flow and functional coverage confirms its utility, particularly in contexts where a richer initial test suite is already in place.

5.7.4 Hybrid configuration 2: first Claude, then genetic

General overview

In the second hybrid configuration, the testbench generated by Claude 3.7 serves as the starting point for genetic amplification. This initial suite already includes 3237 test cases, more than double the original baseline, exhibiting strong initial coverage metrics, especially in function and branch dimensions.

From this enriched foundation, the genetic search proceeds to explore further behavioural diversity through targeted mutation and crossover. In total, the evolutionary process generated 33826 new test cases. Among these, 5112 were discarded due to runtime exceptions, resulting in a pass rate of 84.89% and a fail rate of 15.11%. Compared to the earlier genetic-only experiment, this represents a slightly lower success ratio, likely due to the increased complexity of Claude's tests, which often have deeply nested test structures. This can make mutation tricky to execute successfully.

Table 5.22: Coverage improvement from Claude 3.7 to Hybrid 2 (Claude → Genetic)

Coverage Metric	Baseline (%)	Claude 3.7 (%)	Hybrid 2 (%)	Δ Claude vs Hybrid 2
Statement Coverage	62.26	71.12	71.79	+0.67
Branch Coverage	50.49	67.19	70.38	+3.19
Function Coverage	76.99	84.77	85.06	+0.29
Line Coverage	69.27	77.38	77.94	+0.56

Table 5.23: Coverage progression across generations (Gen 0–5)

Coverage Metric	Base	Gen 1	Gen 2	Gen 3	Gen 4	Gen 5
Statement	71.12%	71.33%	71.50%	71.66%	71.89%	71.79%
Branch	67.19%	68.34%	69.10%	69.75%	70.19%	70.38%
Function	84.77%	84.88%	84.98%	85.03%	85.03%	85.06%
Line	77.38%	77.51%	77.65%	77.81%	77.89%	77.94%

Nevertheless, the combined approach significantly expands the behavioural space under test, with a test inflation factor of 1045.45% when compared to Claude's testbench. This scale of amplification suggests that while Claude provides strong initial coverage through analytical generation, the subsequent application of evolutionary search enables broader structural probing.

Coverage results

Coverage outcomes for the second hybrid configuration, where Claude 3.7 is applied first, followed by genetic search, are summarised in Table 5.22, alongside the individual results of Claude alone for comparison. This side-by-side view helps isolate the contribution of the genetic algorithm when applied to an already high-performing, LLM-generated test suite.

Branch coverage shows the most notable improvement over Claude alone, with a further increase of +3.19 percentage points, indicating that the genetic phase is particularly effective in uncovering new conditional paths beyond what was reached via static LLM-based generation. While gains in statement, function, and line coverage are small, ranging from +0.29% to +0.67%, they still reflect some meaningful additions to the structural reach of the test suite. The search process stopped at an even lower generation than before, at generation 5. This is again based on the stopping criteria. This time, the increase in coverage between generation 4 and 5 was only 0.19%, which is below the 0.25% threshold to stop the search.

Importantly, these results highlight the complementary strengths of the two methods. The fact that Claude has access to the source code enables it to identify and test complex, untested functions that other LLMs or manual efforts failed to cover. The hybrid process then further expands this reach, not by targeting entirely new functions, but by enhancing path diversity within the functions already discovered.

A small remark that has to be made here, is that the initial testbench in this hybrid pipeline was generated by GPT-4o, then augmented by Gemini 2.0, and subsequently amplified using Claude 3.7 before the genetic algorithm was applied. This layered setup means that by the time genetic search begins, almost all coverable functions are likely already exposed. As such, only a very limited increase in function coverage can be expected from the genetic phase. However, the observed rise in branch coverage, though small, is still meaningful, reflecting the inherent strength of mutation and crossover in refining logical path exploration. These findings contribute to the fact that the genetic search stopped very early, at generation 5.

5.8 Claude's Superior Capabilities

Across all experiments where Claude 3.7 was used, whether in isolation or as part of a hybrid pipeline, it consistently elevated function coverage to levels that no other method achieved. While the initial dataset already demonstrated a strong function coverage baseline of 77%, generated

using Gemini 2.0 with focused augmentation strategies, Claude was able to extend this even further to 84-85%. This 7-8 percentage increase isn't obtained by finding untested functions that are called by tested functions in some rare branches, but rather by successfully generating tests for untested functions. This is clear by the fact that search-based algorithms can't increase the function coverage with such a large percentage. These functions include private helpers, tightly nested conditional structures, and other elements that are notoriously difficult to access in Solidity-based contracts due to their complexity.

Even when state-of-the-art LLMs such as GPT-4o, or Gemini 2.0 were employed during dataset generation or in standalone amplification, they were unable to successfully target these difficult segments of code. Manual efforts were also unsuccessful in constructing valid tests for these areas, underscoring the inherent difficulty of the problem. Claude's success in this regard shows its capacity to generate semantically rich, executable test cases that fulfill the intricate preconditions of these complex functions. This finding reinforces Claude 3.7's superiority in this context, especially when function-level completeness is a priority.

5.9 Hybrid evaluation

In addition to evaluating the standalone performance of genetic search and Claude 3.7, two hybrid strategies are considered. These hybrid configurations aim to incrementally build the test suite by applying both techniques in sequence. The test suite generated by the first method serves as the input for the second, allowing it to amplify or refine areas that may have been missed. This layered approach enables each method to contribute its strengths: semantic reasoning in the case of Claude, and structural exploration in the case of genetic search.

By design, the hybrid configurations cannot perform worse than the technique used first, since they extend and enrich its output. This setup creates a structural advantage for hybrid methods and ensures that, in theory, their coverage results match or exceed those of the individual techniques involved. However, as the results reveal, the actual outcome still depends on the sequencing and the nature of the generated test cases.

The results in Table 5.24 indicate clear performance differences between the individual and hybrid test generation strategies. Among the standalone techniques, Claude 3.7 consistently achieves higher coverage than genetic search across all evaluated metrics: statement, branch, function, and line coverage. This confirms the effectiveness of language model-based amplification in identifying complex control paths and exercising semantic aspects of the code that are difficult to reach using traditional evolutionary approaches. While this evaluation couldn't be made on the previous dataset, it can be made here but the outcome remains the same. LLM-based amplification methods outperform search-based methods by quite a big margin.

Hybrid strategies, which combine both techniques sequentially, show varying degrees of success depending on the order in which the methods are applied. Hybrid 2, which applies Claude 3.7 first followed by genetic search, achieves the highest overall coverage. This sequencing allows the LLM to create a strong, semantically rich foundation, which the genetic algorithm then amplifies by exploring additional structural and control flow variations. The result is a consistently higher coverage across all metrics, with particularly strong gains in branch and function coverage. This makes sense since the crossover operation from genetic search thrives from a rich initial dataset.

Table 5.24: Coverage comparison across strategies and hybrid approaches

Coverage Metric	Baseline (%)	Genetic Search (%)	Claude 3.7 (%)	Hybrid 1 (%)	Hybrid 2 (%)
Statement	62.26	65.23	71.12	69.81	71.79
Branch	50.49	57.44	67.19	64.05	70.38
Function	76.99	77.90	84.77	84.26	85.06
Line	69.27	71.75	77.38	76.23	77.94

In contrast, Hybrid 1, which applies genetic search first and then Claude 3.7, does not outperform the standalone use of Claude. In fact, it leads to slight decreases in all coverage metrics. This suggests that the quality and structure of the test suite passed from the genetic phase may limit the effectiveness of the LLM's amplification, due to redundancy and noise in the initial input. Despite applying two techniques, the resulting coverage does not improve, highlighting that hybridisation alone does not guarantee better results unless the sequencing is well aligned with the strengths of each method. These findings emphasise two key insights.

- The order in which techniques are applied has a significant impact on the final coverage.
- Hybrid strategies can lead to substantial gains, but only when the initial phase produces a high-quality, semantically meaningful test suite.

While the general comparison outlines the overall performance trends across all strategies, a more detailed inspection reveals several noteworthy dynamics. Although the full set of results is already presented, certain comparisons offer additional insight into how and why performance differences emerge. These focused comparisons are not just about numerical differences, but about what those differences reveal regarding the interaction between semantic and structural test generation methods. Below are two interesting observations that can be made when looking at the raw numbers.

Observation 1: Claude 3.7 vs Hybrid 1 (Genetic → Claude): An unexpected outcome emerges when comparing Claude 3.7 to Hybrid 1, which applies genetic search first and then Claude. This configuration performs slightly worse than Claude alone across all coverage metrics.

This result highlights a key limitation: when Claude is applied to the output of genetic search, it receives a large, token-heavy test suite with many redundant cases. Despite efforts to filter and batch these inputs, the resulting test files often constrain Claude's generative capacity. Due to token limits, Claude produces fewer new tests per file, struggling to distinguish between covered and uncovered paths within the noisy input.

Moreover, unlike in Hybrid 2, where Claude starts from scratch with a clean and semantically rich baseline, here it must refine structurally mutated tests that lack novelty. This reduces the effectiveness of its amplification and shows that the quality of the input, and its size, matters as much as the technique itself.

This comparison reinforces a broader point: Hybridisation does not guarantee improvement. If the first stage produces noisy or bloated input, it can hinder rather than help the second. For LLM-based techniques like Claude, starting from a compact, meaningful test suite is critical for maximising coverage gains.

Observation 2: Hybrid 1 vs Hybrid 2: The final comparison focuses on the two hybrid strategies that combine genetic search and Claude 3.7, differing only in the order of application. Results show that Hybrid 2 (Claude → Genetic) consistently outperforms Hybrid 1 (Genetic → Claude) across all coverage metrics. The improvement is most notable in branch coverage (+6.33%), but also extends to statement, line, and function coverage, just more modestly.

This outcome aligns with the observed strengths of each technique. Claude generates the most effective results when it starts from a clean, semantically coherent base. In Hybrid 2, it produces a high-quality initial test suite that genetic search can further expand and diversify without being constrained by token limitations. This is not the case in Hybrid 1, which is why it perform worse. This sequence enables meaningful amplification, particularly in structurally complex regions such as alternative branches.

These findings highlight a key insight: the sequence of techniques matters. Claude excels in providing semantic depth, while genetic search contributes structural diversity. When Claude comes first, the strengths of both methods are preserved and combined effectively. The reverse order, however, introduces noise and redundancy that limit the overall impact.

In summary, Hybrid 2 achieves higher coverage and uses both techniques more efficiently. This demonstrates the importance of thoughtful sequencing in hybrid test generation and offers practical guidance for designing future amplification strategies.

5.10 Anecdotal Evidence

While quantitative metrics such as coverage percentages offer a valuable overview of each technique's effectiveness, they often conceal important contract-specific variations. Throughout the experiments, it became evident that performance is not uniformly distributed across all contracts. In particular, some contracts showed substantial coverage improvements when subjected to search-based techniques, yet remained largely unaffected by LLM-driven approaches. Conversely, other contracts benefited considerably more from LLM-based approaches, with minimal gains observed from search-based methods. Finally, a subset of contracts appeared resistant to both strategies, yielding negligible coverage improvements regardless of the method applied.

This section aims to manually investigate these outliers in order to identify the structural, syntactic, or semantic characteristics that may explain the observed differences. Understanding these patterns can offer insight into the strengths and limitations of each approach and may guide future efforts in selecting or designing test generation strategies tailored to specific contract types.

5.10.1 Manual Analysis: Genetic Search Excelling in Highly Constrained Contracts

Upon manual inspection of individual contract files, a recurring pattern emerges in which genetic search outperforms LLM-based techniques. This is particularly visible in highly constrained contracts, characterised by the frequent use of assertive guard conditions, typically implemented through require statements in Solidity, or in some cases, by using regular if-statements where the if-body throws a 'revert' exception. There are contracts that enforce strict preconditions and postconditions for nearly every function, introducing multiple distinct branches that must be explicitly tested.

Each 'require'-statement or if-statement creates a bifurcation in control flow: one where the condition holds and execution continues, and one where the assertion fails, resulting in transaction reversion. To achieve full branch coverage, a testing strategy must account for both execution paths, that is, the path where the condition passes and the one where it fails. This demands an extensive and nuanced test suite that not only reaches these conditions but also handles the expected failure cases correctly.

Genetic search is particularly suited to this challenge due to its inherently exploratory nature. Through random mutations and crossover operations, it generates a large and diverse volume of test cases. Over successive generations, these tests evolve to explore various execution paths, including both successful and failing branches. Crucially, the genetic process can randomly assign expected outcomes, choosing whether a test should expect success or failure, enabling it to naturally discover and retain tests that exercise both sides of each 'require'-statement.

In contrast, LLM-based approaches face limitations in this context. While they are capable of producing well-structured tests, the limited output window imposed by token constraints makes it difficult for an LLM to address every precondition and postcondition in a highly guarded contract. If each function contains several assertions and the contract itself spans many functions, a single amplification prompt may be insufficient to provide adequate test coverage across all branches. As a result, some critical paths remain untested, particularly those involving failure states, which are less intuitive to cover and often omitted unless explicitly prompted.

This analysis suggests that in scenarios where contracts are densely populated with assertive checks, search-based methods like genetic search are structurally better positioned to achieve high coverage. Their ability to rapidly generate a large, varied pool of test inputs makes them more effective at exploring the complex branching logic imposed by assertions. LLMs are powerful

in understanding code semantics but are inherently limited by the volume of content they can produce in a single prompt. Unfortunately, this restricts their effectiveness in highly constrained environments.

Example

To test the transferFrom method in Figure 5.1, and obtain high branch coverage, one needs to test all four preconditions separately (four tests to test the failures of the requirements), and the if-statements at the bottom of the function (at minimum an additional three tests). To conclude, the most efficient way to obtain 100% branch coverage for just this one function, would require at least seven tests. Not only is this a lot of tests for an LLM to produce for a single function (in the case of one prompt), but setting up the state of the contract in such a way that each of the preconditions fail when they are supposed to, is a complex task. Search-based methods have less problems with this because the odds of eventually covering such a path, is more likely to happen due to the larger number of outputted tests. In the coverage report, a black box with E means the 'else' path was not taken, and a black box with I means the 'if' path was not taken. This indicates which branches of the code were not executed during testing.

```
function transferFrom(address _from, address _to, uint256 _amount) returns (bool success) {
  E require(!locked);
  require(balanceOf[_from] >= _amount);
  E require(balanceOf[_to] + _amount >= balanceOf[_to]);
  E require(_amount <= allowance[_from][msg.sender]);
  balanceOf[_from] -= _amount;
  uint256 preBalance = balanceOf[_to];
  balanceOf[_to] += _amount;
  allowance[_from][msg.sender] -= _amount;
  bool alreadyMax = preBalance >= singleIDXMQty;
  E if (!alreadyMax) {
    E if (now >= validAfter[_to].ts + mustHoldFor) validAfter[_to].last = preBalance;
    validAfter[_to].ts = now;
  }
  I if (validAfter[_from].last > balanceOf[_from]) validAfter[_from].last = balanceOf[_from];
  Transfer(_from, _to, _amount);
  return true;
}
```

Figure 5.1: Example 1: highly constrained function

5.10.2 Manual Analysis: LLMs Excelling in Semantically Specific or Privileged Behaviour

Conversely, manual inspection reveals that LLM-based approaches outperform search-based algorithms in contract scenarios where semantically specific behaviour is present. These are typically contracts with strict privilege checks, hardcoded parameter dependencies, or tightly scoped business rules, all of which require precise knowledge of the internal contract state in order to trigger specific execution paths.

A common pattern involves functions restricted to the contract's owner address. This owner-only access control is often enforced by using the 'onlyOwner' keyword outside the function body, and cannot be easily satisfied through randomised input generation. In search-based methods like genetic search, unless the input test data happens to mutate the sender address to match the owner's address, these functions will not execute successfully, making the associated code paths inaccessible. While these mutations can happen, they are rather rare because during testing, these mutations often caused more failing tests, rather than passing ones, so the weight for these mutations is low.

Additionally, functions involving dynamic parameters such as transaction "taxes" or gas fees present another challenge. In Solidity, these may be implemented as private state variables (e.g. a

gasPrice set to a fixed rate such as 4.5%). If this value is not explicitly referenced in existing tests, genetic algorithms have no guidance in discovering the correct input to trigger these branches, or may not even know of their existence. Random mutation of numerical parameters would be required to guess, for instance, a float of exactly 0.045, which is statistically improbable without prior context. This makes many execution paths invisible to the search algorithm.

In contrast, LLMs perform well under these constraints because they can access and use the contract's source code, unlike search-based techniques that depend solely on input-output relations and mutation heuristics. LLMs can read internal variables, infer constraints, and directly incorporate them into test generation. For example, if a function depends on gasPrice being 4.5%, the LLM can identify this value in the contract's code and explicitly construct a passing test using that exact parameter. Similarly, if access is limited to a specific address, LLMs can identify that address and write appropriate tests that simulate the correct sender identity. This allows LLMs to cover branches that depend on very specific conditions.

Example

To test the feeFor method in Figure 5.2, one needs to understand how the fee is computed, in order to verify whether certain inputs yield the output fee. The LLM will have the advantage because it can see in the documentation how the calculation must be done, and can make tests based on this, while search-based methods do not have access to this, and have to 'guess' what the output might be.

```
/* --------------- fee calculation method ---------------- */

/**
 * @notice 'Returns the fee for a transfer from `from` to `to` on an amount `amount`.
 *
 * Fee's consist of a possible
 *    - import fee on transfers to an address
 *    - export fee on transfers from an address
 * IDXM ownership on an address
 *    - reduces fee on a transfer from this address to an import fee-ed address
 *    - reduces the fee on a transfer to this address from an export fee-ed address
 * IDXM discount does not work for addresses that have an import fee or export fee set up against them.
 *
 * IDXM discount goes up to 100%
 *
 * @param from From address
 * @param to To address
 * @param amount Amount for which fee needs to be calculated.
 *
 */
function feeFor(address from, address to, uint256 amount) constant external returns (uint256 value) {
  uint256 fee = exportFee[from];
  if (fee == 0) return 0;
  uint256 amountHeld;
  if (balanceOf[to] != 0) {
    if (validAfter[to].ts + mustHoldFor < now) amountHeld = balanceOf[to];
    else amountHeld = validAfter[to].last;
    if (amountHeld >= singleIDXMQty) return 0;
    return amount*fee*(singleIDXMQty - amountHeld) / feeDivisor;
  } else return amount*fee / baseFeeDivisor;
}
```

Figure 5.2: Example 2: complex function

5.10.3 Contracts Where Coverage Does Not Improve

While both LLM-based and search-based testing methods have demonstrated the ability to improve code coverage in many contracts, there remains a subset of contracts where no observable increase in coverage metrics occurs after applying these techniques. Manual inspection reveals that three main factors contribute to this stagnation.

First, some contracts already exhibit high coverage from the baseline test suite. In these cases, the existing tests may already exercise nearly all relevant control paths, leaving little room for further improvement. After all, certain contracts have very little assertions or branches, and simply perform sequential operations without too many checks. When the initial test coverage is close to saturation, neither additional search-based mutations nor LLM-generated cases provide meaningful gains, simply because the remaining branches are trivial.

Second, in rare cases, certain contracts contain unreachable code. These are code paths that cannot be activated under any practical input configuration due to internal contradictions or unresolvable conditionals. Coverage tools may still report these lines as uncovered, but no automated method, whether mutation-based or prompt-driven, can access them unless the code is explicitly altered. As a result, coverage metrics remain unchanged despite the introduction of new tests.

Lastly, several contracts are characterised by deep implicit preconditions that are not immediately visible in the function signature or external interface. For example, a function may require the contract to be in a specific internal state or assume that other functions have been called previously to set up certain requirements. These conditions are often not captured by the baseline tests and are difficult for genetic search to infer due to their specificity. Similarly, LLMs may miss them if they are not explicitly documented in the structure of the code. As a result, neither method is able to generate tests that satisfy these hidden constraints, leading to no further coverage gains.

Example

To test the burnFrom method in Figure 5.3, one needs to call the method, and then access the getters for the balances and allowance afterwards, in order to see whether the operations happened successfully. However, by simply calling this method once, you already cover all lines and all branches except for one. So additional testing can be useful to test the correctness of the function, but will not gain increases in coverage metrics.

```
function burnFrom(address _from, uint256 _value) public returns (bool success) {
    E require(balances[_from] >= _value);                // Check if the targeted balance is enough
    require(_value <= allowance[_from][msg.sender]);    // Check allowance
    balances[_from] -= _value;                          // Subtract from the targeted balance
    allowance[_from][msg.sender] -= _value;             // Subtract from the sender's allowance
    totalSupply -= _value;                              // Update totalSupply
    emit Burn(_from, _value);
    return true;
}
```

Figure 5.3: Example 3: basic sequential function

5.11 Faults in the Contracts

While the primary objective of this study is to compare the effectiveness of LLM-based and search-based test amplification methods, an important observation emerged during the evaluation process: several contracts in the dataset appear to exhibit genuine functional faults. This observation is not incidental but stems from the nature of the dataset itself (see Section 4.4.4 about dataset generation), which consists of smart contracts previously flagged as 'potentially vulnerable'. These contracts have typically not undergone thorough prior testing, and some are known to contain certain vulnerabilities and faults. However, the makers of the dataset do not disclose where the vulnerabilities occur in the contracts.

As part of the evaluation, generated test cases, particularly from LLM-based approaches, frequently fail when executed. Although the majority of these failures can be attributed to syntactic errors or incorrect test assumptions, a smaller subset of failures appears to reflect unexpected contract behaviour. Upon closer inspection, these tests are syntactically and semantically correct

but produce failing assertions due to incorrect internal logic in the contract. For instance, one example involves a buy function where, rather than transferring funds to the seller as expected, the contract mistakenly deducts funds from both the buyer and the seller. In such cases, the generated test case rightly asserts that the seller's balance should increase, and the test fails when this does not occur. This failure, rather than indicating an error in the test logic, highlights a flaw within the contract itself.

It is important to clarify that these observations are not the result of artificial faults introduced through mutation testing. Rather, they are inherent to the original contracts and represent real-world vulnerabilities. While this section does not aim to evaluate fault detection capabilities of test amplification techniques directly, these instances of real vulnerabilities offer valuable supplementary insights to the research.

During the evaluation of the results, failing test cases are not discarded, in contrast to the workflow adopted in Meta's TestGen-LLM system [1], where non-passing cases are filtered out early to ensure measurable improvements. While such filtering helps maintain a high-quality test suite, it may also eliminate cases that offer valuable diagnostic insights. Many failures stem from syntactic issues or incorrect assumptions. However, a subset of failing tests does not fit these categories and may point to deeper issues, such as unexpected contract behaviour. Rather than treating these cases as noise, they are considered as potential sources of verification. Examining them more closely may contribute to understanding both the limitations of the test generation process and the underlying robustness of the contracts under study. This approach highlights the added value of retaining and then further analysing failures of the tests, rather than discarding them by default.

5.11.1 Detected faults

The following subsection presents a selection of faults identified during manual inspection of failing test cases that could not be attributed to syntactic errors or incorrect assumptions. These failures originated from semantically valid tests, suggesting incorrect contract behaviour. By analysing these cases in detail, it becomes possible to highlight specific flaws in contract logic that were surfaced through the test amplification process. Each example listed here reflects an instance where the contract's implementation deviates from its intended functionality, offering insight into the type of faults that may remain undetected without systematic testing.

- **Incorrect comparison logic in balance and allowance checks**. Several contracts perform logic errors in key conditional checks. For instance, one contract checks whether the transfer value is greater than the available balance or allowance, and proceeds if this is the case. This is logically inverted: the condition should ensure that the balance or allowance is greater than or equal to the value before allowing the transaction. Such an inversion leads to both valid and invalid transactions failing, which undermines expected contract functionality.
- **Unrestricted or unvalidated token transfers**
  - A token distribution function is implemented without validating whether the source account has sufficient balance to cover the distributed amount. This allows tokens to be distributed even when the source has none, effectively enabling arbitrary token creation.
  - In other transfer-related functions, there is no check to ensure sufficient balance exists before performing deductions. As a result, transactions subtract the specified amount from both the sender and receiver, potentially leading to negative balances for both accounts, a critical error which can lead to financial loss.
  - A related vulnerability involves a comparison using only '>' instead of '>=', which prohibits users from spending their full balance. If a user has exactly 100 tokens and tries to spend all 100, the transaction fails, despite being valid.

- **Missing updates to crucial state variables**. One minting function increases balances but fails to update the totalSupply. This discrepancy between the internal state and the actual circulating tokens breaks certain assumptions. Certain functions rely on the totalSupply number and this error will make it so all results from those functions have the possibility of being invalid.
- **Solidity version-specific parameter handling**. A contract written in Solidity 0.4.19 fails to mark string parameters as memory. In this version, omitting memory causes the parameter to be stored in storage, which is not permitted for function parameters and leads to misinterpreted values or silent errors. For example, the setter that sets the string value, is unable to do so due to this error. This type mismatch can severely impact the data consistency.
- **Missing critical event emission**. Two functions that perform nearly identical actions differ in a significant aspect: only one of them emits a required event upon execution. In systems relying on event logs for external tracking, the absence of this event can lead to a loss of visibility and potentially exploitable blind spots.
- **Incorrect access control and isolated contract instances**. In certain tests, contracts are deployed independently rather than as linked components. This leads to failures when certain functions are restricted to the owner, and no valid ownership context exists in the isolated deployment. Moreover, preliminary inspection of the logic within such functions suggests the presence of communication errors (between deployments), indicating further vulnerability risks if contracts are deployed independently.
- **Ambiguous ownership and transfer behaviour**. In one contract, ownership tests fail because no valid address is assigned as the owner at deployment. Additionally, test results suggest inconsistent handling of initial balances and ownership-related roles, leading to test failures and unclear state transitions.

Summary

The discovery of real-world vulnerabilities during test amplification highlights a noteworthy side-effect of the evaluation process. In particular, LLM-based methods occasionally surface deeper contract issues, not by explicitly targeting vulnerabilities, but by generating plausible "happy path" tests that fail despite being logically and syntactically valid. These failures signal potential flaws in the contract's logic that only become evident upon manual inspection. This underscores the added diagnostic value of LLM-generated test cases: while not explicitly aimed at fault detection, they can inadvertently reveal contract misbehaviour through unexpected assertion failures.

Search-based techniques, while capable of reaching similar execution paths, typically lack the semantic cues, such as meaningful variable names or intent-revealing comments, that help identify when a behaviour is incorrect. As a result, their generated tests may still trigger faulty behaviour, but without clear assertions or context, such failures may go unnoticed or unrecognised as indicators of underlying vulnerabilities.

This suggests that LLM-based test amplification, when combined with manual follow-up, can support the discovery of latent bugs or incorrect logic in poorly tested contracts. While this is not the primary goal of amplification, or of the study in general, it provides an additional benefit worth considering. It also justifies the choice to retain and inspect failing test cases rather than discarding them outright, as these failures may hold valuable insights into the contract's correctness.

Chapter 6

Threats to Validity

This section discusses potential threats to the validity of the evaluation results, structured into the four standard categories: construct validity, internal validity, external validity, and reliability.

6.1 Construct Validity

Construct validity concerns whether the evaluation truly measures what it intends to measure.

6.1.1 Compiler-generated functions affecting coverage metrics

Solidity implicitly generates certain functions, most notably getter functions for public state variables, that are not written in the contract source code but are added during compilation. These functions become externally accessible and are included in the compiled interface of the contract, meaning they are visible to external callers and test tools. As a result, coverage tools include them in function coverage calculations, even though they are not explicitly defined or visible in the contract source.

This can lead to skewed coverage metrics. For example, if a contract defines three public variables and three functions, the compiled contract will expose six functions in total, three written by the developer and three automatically generated getters. If tests only exercise the developer-defined functions, the only ones visible in the source code, the function coverage will appear as 50%, even though all intended logic may be covered.

This issue affects the comparison between LLM-based (white-box) and search-based (black-box) amplification. An LLM may generate tests that call these implicit getter functions, increasing apparent coverage. A search-based approach, lacking access to source structure, may not trigger these calls unless they happen to be used in existing tests. This may introduce a bias in coverage-based evaluation, favouring LLM-based methods without reflecting a true difference in test effectiveness.

6.1.2 Fitness function granularity

The fitness function used in the genetic search is based solely on code coverage metrics (e.g. line or function coverage), without considering the semantic quality of the generated tests. As a result, the search may produce tests that trigger new code but lack meaningful assertions or validation logic. These shallow tests can inflate coverage numbers without providing corresponding improvements in mutation detection or test quality. This limits the interpretability of the coverage gains as a true measure of test effectiveness. While small-scale mutation testing was performed in the results to get a sense of fault detection, it was not incorporated into the fitness function. Furthermore, due to the limited scope of this mutation analysis, it was difficult to fully assess the true effectiveness of the generated tests beyond coverage alone.

6.2 Internal Validity

Internal validity relates to whether the observed results are caused by the techniques under study, rather than other unknown factors.

6.2.1 Coverage ceiling

Search-based test amplification operates as a black-box technique and is therefore limited by the behaviour exposed in the original test suite. If certain functions or paths are not reachable from existing tests, the algorithm may converge prematurely on local optima, achieving submaximal but seemingly stable coverage. In this study, the observed coverage improvements (e.g. from 60% to 70%) are interpreted as absolute gains, but the true theoretical maximum may be slightly higher (e.g. 72%), meaning the relative improvement could be more significant than reported. While this limitation is discussed in the context of comparing search-based and LLM-based methods, particularly when analysing differences in accessible code paths, it remains a threat to validity in interpreting raw coverage gains. Without explicitly modeling or estimating the coverage ceiling per contract, the reported results may understate or overstate the practical effectiveness of each method.

Importantly, LLMs also face coverage ceilings, although they can potentially raise them by generating tests for previously untouched contract functions. However, the presence of helper functions or utilities in the same file, such as libraries or unused internal logic, can artificially inflate the number of uncovered elements, lowering the apparent ceiling. Since such functions are not always exercised through any contract path, this can lead to an underestimation of amplification effectiveness, especially when interpreting raw coverage values without filtering for relevance.

6.2.2 Dependence on initial seed tests

The effectiveness of both search-based and LLM-based amplification strategies is strongly influenced by the structure and quality of the initial test suite. For genetic search, operations like crossover depend on having a diverse pool of tests; if the initial tests are too simplistic or homogeneous, the algorithm may struggle to produce deeper or more varied test behaviour. Similarly, LLM-based amplification also exhibits sensitivity to the input test suite. This was especially evident in the Hybrid 1 configuration, where the LLM was tasked with amplifying an already enlarged test suite produced by genetic search. The excessive size and redundancy of the test suite appeared to hinder the LLM's ability to generate useful additions, resulting in lower performance compared to applying the LLM directly to the original base tests. These observations highlight that both approaches are not immune to the limitations or noise introduced by their starting point, and that the initial test context can significantly affect amplification outcomes.

6.3 External Validity

External validity addresses the generalisability of the findings beyond the studied setting.

6.3.1 Limited input space encoding

The effectiveness of search-based test amplification is influenced by the diversity and precision of the input mutations it applies. In this study, mutations are limited to a few basic data types: addresses, strings, and numeric values, which are treated generically without distinction between specific Solidity types (e.g. uint8, uint256, int). This coarse-grained mutation strategy may overlook edge cases or type-specific behaviours relevant to smart contract logic, such as overflow conditions, boundary values, or input constraints enforced by specific type annotations.

As a result, the search space explored may be narrower than intended, potentially limiting the coverage or fault detection capacity of the amplified tests.

6.3.2 Artificial absence of import statements

In the dataset used throughout this research, all Solidity contracts are entirely self-contained: any inheritance structures or external library usages (e.g., SafeMath) are embedded directly within the contract files themselves. As a result, no import statements are present. While this design simplifies test generation and avoids dependency resolution issues, it does not fully reflect the structure of real-world smart contract projects, where reuse through imports is the norm. This limits external validity, especially for integration or system-level tests. However, since this work focuses solely on unit-level test amplification, where contracts are often compiled and tested in isolation, the impact of this limitation is considered acceptable. Nonetheless, this is a limitation of the dataset and can limit the capabilities of doing test amplification higher up in the V-model, such as test amplification for integration tests.

6.3.3 Dual use of LLMs in testing

This small-scale study investigates whether using the same large language model for both test generation and test amplification affects the resulting code coverage. Specifically, it examines whether self-amplification, where generation and amplification are performed by the same model, yields different results than cross-amplification, in which different models are used for each step. Two LLMs are evaluated: ChatGPT 4o and Claude 3.5. The study focuses exclusively on standard code coverage metrics: statements, branches, functions, and lines. While these experiments offer initial insights, the results from this limited set as well as from the broader test bench should be interpreted with caution. The observed effects may not generalise across other contracts, models, or testing contexts, and further evaluation is necessary to confirm their broader applicability.

The evaluation is conducted on a small subset of benchmarks, specifically the first 10 contracts from the Contracts Test Bench 1 suite. For each contract, an initial test suite is generated using one of the LLMs. Then, amplification is applied using either the same model (self-amplification) or the other model (cross-amplification). Results can be found in Table 6.1.

The following four configurations are tested:

- ChatGPT 4o generation → Claude 3.5 amplification
- ChatGPT 4o generation → ChatGPT 4o amplification
- Claude 3.5 generation → Claude 3.5 amplification
- Claude 3.5 generation → ChatGPT 4o amplification

Key Observations

Coverage increases across all configurations: All setups result in notable improvements in coverage metrics after amplification. This confirms the general utility of LLM-based amplification techniques, regardless of the pairing.
Cross-amplification yields higher coverage than self-amplification: Using a different model for amplification than the one used for generation generally results in higher final coverage. For example, with ChatGPT generation, statement coverage increases from 56.64% to 74.07% when amplified by Claude, compared to 69.72% with ChatGPT amplification. Another case is where branch coverage follows a similar pattern, improving from 40.09% to 57.83% with Claude amplification, versus 52.76% with ChatGPT.

The trend is similar when starting from Claude-generated tests. Amplification with Claude results in slightly higher coverage (e.g., function coverage: 82.07%) than amplification with ChatGPT (79.89%), though the gap is narrower in this direction.

Self-amplification leads to more conservative gains: In both directions, self-amplification consistently leads to slightly lower coverage increases than cross-model setups. This may suggest that using a different model for the second step is more likely to introduce test logic or inputs that complement the initial generation strategy, leading to broader coverage.

Claude-generated tests start from a higher baseline: Initial coverage values are higher for Claude 3.5 than for ChatGPT 4o. For example, Claude-generated suites start with 62.96% statement coverage versus 56.64% for ChatGPT. This reduces the potential margin for improvement during amplification, though the relative trend remains consistent.

To conclude, In this targeted study on the first 10 contracts of the Contracts Test Bench 1, cross-amplification using a different LLM for amplification than for generation results in higher code coverage than self-amplification. While all approaches yield improvements, using a second model during amplification appears to offer a complementary perspective that enhances test diversity. However, due to the limited scope of this evaluation, further large-scale studies are needed to determine whether these patterns generalise across broader sets of contracts, LLMs, and domains.

6.4 Reliability

Reliability relates to the consistency of the results when reproduced under similar conditions.

6.4.1 Tool and Model Dependence

Results depend on specific tools: the version and behaviour of the LLMs (ChatGPT 4o and Claude 3.5), the coverage instrumentation, and the mutation operators used in search-based amplification. Changes in these components, such as different prompts, model versions, or instrumentation libraries, could influence both the quality of generated tests and the resulting coverage. Moreover, while the test generation and amplification were performed using a scripted pipeline, certain prompts or decisions during evaluation (e.g., mutation types, runtime limits) involve researcher choice, which may impact reproducibility.

Table 6.1: Amplification results across generation-amplification pairings

Metric	**4o → Claude 3.5**	**4o → 4o**	**Claude 3.5 → Claude 3.5**	**Claude 3.5 → 4o**
Statement Coverage (%)				
Original	56.64	56.64	62.96	62.96
New	74.07	69.72	71.90	71.02
Branch Coverage (%)				
Original	40.09	40.09	40.32	40.32
New	57.83	52.76	57.60	58.53
Function Coverage (%)				
Original	71.20	71.20	75.54	75.54
New	85.87	82.61	82.07	79.89
Line Coverage (%)				
Original	63.13	63.13	69.78	69.78
New	77.20	74.05	76.11	75.00
Passing/Failing Tests				
Original	82/0	82/0	91/0	91/0
New	155/24	131/37	133/50	126/62

6.4.2 Token Limit Uncertainty in GitHub Copilot Pro

An additional reliability threat arises from the undocumented and potentially restrictive token limits applied by GitHub Copilot Pro. Although the underlying GPT-4o model supports extended context windows of up to 128k tokens via the API, Copilot's implementation within Visual Studio Code may enforce significantly lower effective limits. Informal reports, including those on StackOverflow and GitHub Discussions, suggest a practical ceiling near 4000 tokens, which may truncate model input and affect output quality. [110, 111]

This uncertainty is critical for LLM-based test amplification. A smaller context window can prevent the model from fully ingesting a contract or its associated test suite, resulting in incomplete or less relevant test cases. Such limitations are especially problematic in Solidity, where contract logic often relies on interdependent functions and precise test context.

Because GitHub does not officially document these constraints or provide configurability, the amplification results obtained through Copilot Pro may be influenced by infrastructure-level limitations rather than model capability alone. This reduces the reliability and reproducibility of results, particularly when comparing LLMs across platforms or configurations. [111]

Chapter 7

Conclusion

This thesis investigated automated test amplification for Solidity-based smart contracts using both large language models and search-based techniques. The evaluation focused on structural coverage, efficiency, and the potential of hybrid strategies. The research was guided by four core research questions, which are revisited below to synthesise the key findings and insights.

Effectiveness of Test Amplification Methods

LLM-based amplification methods consistently achieved stronger improvements in structural coverage, especially for branch and function coverage. Models like Claude 3.7 Sonnet and ChatGPT 4o generated semantically meaningful and structurally diverse tests, discovering execution paths that were previously untested. Notably, some of these tests exposed failing behaviours tied to potential vulnerabilities or logical flaws in the contracts, highlighting the importance of preserving failing outputs during evaluation.

Search-based methods, particularly genetic search, also improved structural metrics but typically in a more constrained fashion. Their strength lies in iteratively refining inputs to uncover edge cases within already-covered functions. Genetic search was especially effective in contracts featuring dense branching logic or assertions, where mutation-based exploration uncovered paths that LLMs sometimes missed due to output size limits.

Scalability and Efficiency Considerations

LLM-based amplification is resource-intensive, given the computational cost of generating context-aware tests through prompt engineering and output parsing. While Claude 3.7 Sonnet delivered the best structural gains, more lightweight models like o3-mini provided competitive results with lower generation costs and improved efficiency. These models demonstrate that test quality is not solely a function of model size, but also prompt design and workflow.

Search-based approaches, by contrast, scale more easily across larger datasets and require minimal infrastructure. Random and guided variants are lightweight but limited in coverage gains, while genetic search yields better results. However, all search-based methods risk producing large volumes of tests, which can reduce test suite maintainability if not properly managed.

Comparative Strengths of LLM- and Search-Based Approaches

While LLMs generally outperformed search-based methods in raw coverage and semantic depth, the two are not in strict competition. LLMs are especially valuable in understanding privileged logic, business rules, and internal conditions that cannot be inferred purely from structural input mutations. This makes them well-suited for amplifying tests in contracts with intricate access controls or conditional flows.

On the other hand, search-based strategies excel when broad exploration of control flow is required. For example, genetic search can evolve unexpected input values that cause non-obvious

branches to be executed, something that LLMs, bound by token size and prompt limits, may overlook. The approaches are therefore complementary: one semantic, one structural.

Benefits of Hybrid Amplification Strategies

A hybrid strategy combining LLM-based test generation with subsequent search-based refinement proved especially effective. Starting with LLM-generated tests that cover a broad semantic base and then applying genetic search to refine and expand structural reach resulted in the highest observed branch coverage, up to 70% across the entire testset. This pipeline leveraged both the context-awareness of LLMs and the targeted exploration of genetic algorithms, producing both breadth and depth in coverage.

This hybrid workflow suggests a promising direction for future amplification pipelines: modular, layered strategies that combine model insight with evolutionary exploration to maximise test quality.

Final Reflections and Outlook

Beyond the contributions, this thesis also produced a reproducible testbench designed for Solidity-based smart contracts, addressing the lack of comprehensive open-source test suites. While limitations such as the computational cost of mutation testing and model inference were encountered, they were addressed through targeted evaluations.

One critical insight is the sensitivity of amplification outcomes to prompt formulation, model selection, and batching configuration. Even small changes in prompt phrasing or model behaviour significantly affected coverage metrics. A small-scale study explored separating test generation and amplification across different models, suggesting that cross-model amplification can improve diversity, though more research is needed.

Although this study focused on Solidity, its findings apply more broadly. The combination of symbolic reasoning (LLMs) and heuristic exploration (search-based methods) is applicable to test amplification in other programming languages and domains.

In conclusion, LLM-based amplification provides strong semantic guidance and structural gains; search-based methods offer lightweight, effective strategies for exploring edge cases; and hybrid techniques combine their strengths for superior test quality. Together, they form a flexible and powerful toolkit for advancing automated smart contract testing workflows.

Chapter 8

Lessons Learned

Throughout the execution of this thesis, several practical insights emerged that were not foreseen during the initial setup. These lessons, while not always directly related to the central research questions, influenced the direction, scope, and feasibility of the experiments, and are therefore worth documenting.

8.1 Transition from Truffle to Hardhat Due to Tool Incompatibility

The first significant lesson relates to the tooling used in the experimental setup. Initially, Truffle was selected as the testing framework due to its widespread adoption and reliable support for JavaScript-based smart contract testing. This setup enabled initial validation of basic contract behaviour using Truffle 5.4.x. However, during early test amplification experiments, it became clear that Truffle was incompatible with Solidity-Coverage, a critical tool for coverage analysis. Specifically, version mismatches between Truffle and Solidity-Coverage resulted in inaccurate or failed coverage reports, blocking the use of structural coverage as an evaluation metric.

This issue prompted a transition to the Hardhat framework. Hardhat not only resolved the compatibility problems but also provided better support for projects using multiple Solidity compiler versions a common characteristic of real-world contracts. In addition, Hardhat's seamless integration with Solidity-Coverage enabled consistent, accurate measurement of line, statement, branch, and function coverage throughout the remainder of the thesis. In hindsight, starting with Hardhat from the beginning would have reduced unnecessary overhead and debugging time.

8.2 Genetic Algorithms Require Structural Diversity for Effective Crossover

One key observation from the genetic search experiments was the dependence of crossover effectiveness on the diversity of the initial test pool. In the first testbench, where most functions were tested by only one or two cases, early generations showed limited gains from crossover. The lack of variation meant there was little useful material to recombine, and mutation alone drove the early improvements. However, just mutation isn't always sufficient to obtain two meaningful parents. Often times, parent tests would be nearly identical, limiting the usability of the crossover.

In contrast, the second testbench, which was augmented with LLM-generated test cases (notably via Gemini 2.0), contained richer structural variety from the outset. As a result, crossover operations became effective much earlier in the search process. This was reflected in a steeper increase in branch coverage between generations 1 and 2, compared to the more gradual improvements in the earlier setup. This finding highlights that crossover can only accelerate coverage

gains when sufficient genetic diversity is present, an insight that supports combining search-based and LLM-based methods in complementary ways.

8.3 LLMs Struggle With Deeply Guarded Branches

Another insight emerged when analysing branch coverage in highly guarded functions. Each require or if statement introduces a bifurcation in control flow, creating both a success and failure path. To achieve full branch coverage, test cases must account for both paths, including those that are expected to fail.

Genetic algorithms proved well-suited for this challenge. Their randomised nature, including the ability to assign expected failures, allowed them to discover and retain tests for both success and failure branches through natural selection. Conversely, LLM-based amplification approaches struggled in this regard. Their token limits and reliance on prompt-based context made it difficult to generate exhaustive test cases that explore both sides of deeply nested conditions. As a result, even when amplification was successful in terms of syntax and structure, many branches remained uncovered—especially in contracts with complex guards or multiple require conditions.

8.4 Selection Operator in Genetic Search can be Tricky to Implement

A key lesson from implementing genetic search in a Solidity testing environment using Hardhat, is that theoretical designs for fine-grained selection based on individual test contributions do not scale in practice. While the ideal approach would involve assessing each new test case individually for its impact on code coverage, this is computationally infeasible given the size of the dataset and the limitations of the Hardhat coverage tool, which does not link coverage metrics to individual test functions. Alternative approaches can be made by manually identifying uncovered branches, and checking whether these branches are covered within compiled contracts at runtime. This was infeasible and lay outside the scope.

To address this, a batching strategy was adopted. Rather than measuring coverage per test, newly generated tests were grouped into four equal-sized batches per contract. Each batch was evaluated separately for its collective impact on coverage. If a batch resulted in an increase—even if only one test was responsible—the entire batch was retained. This significantly reduced the number of coverage analysis runs while still enabling some level of selection pressure.

This compromise illustrates a broader lesson: in realistic testing scenarios, practical limitations often require approximations or heuristic-based solutions. While the batching approach may retain some non-contributing tests, it ensures the scalability of the evaluation process and aligns the algorithm more closely with practical time constraints.

8.5 Normalisation Methods to grant Fairness

After evaluating the original results, it became apparent that a direct comparison between search-based and LLM-based methods is only fair if the playing field is the same for both. LLM can generate tests for previously untested functions, while search-based methods can't do this directly. Since this wasn't implemented to stick to the goal of amplification, rather than generation, a solution had to be found. The decision to make a new testbench was ultimately selected as a solution to this problem. However, previous to that, several normalisation techniques were tested to assess whether normalisation would be able to even the playing field. The following three techniques below were selected and tested, but each of them led to fairness to either LLMs, or search-based methods.

8.5.1 Normalisation Based on Theoretical Maximum Coverage

One proposed method to address the disparity between search-based and LLM-based testing is to normalise coverage increases relative to the theoretical maximum coverage achievable through amplification. This approach attempts to account for the fact that search-based techniques are inherently limited to exploring only those functions that are already referenced in the existing test suite. Consequently, any untested functions that are not reachable through mutation of existing tests remain inaccessible to such algorithms, thus constraining their maximum achievable coverage.

In theory, the maximum achievable coverage could be computed directly from the full coverage report JSON. However, in practice, determining this value with accuracy is problematic. Some untested functions may be invoked through previously unexplored execution paths within already-covered functions. For instance, mutations in the input space might activate conditions that lead to calls to new functions, increasing function coverage without the need for direct test generation. This indirect reachability complicates the estimation of the true upper bound on achievable coverage, making any such normalisation only a rough approximation.

Moreover, applying this normalisation by dividing the achieved coverage increase by the estimated maximum does not fully resolve the fairness issue. For example, if a search-based algorithm improves branch coverage from 65% to 70% in a scenario where the theoretical maximum is 75%, the normalised score would be 6.67% (= 5/75). An LLM that improves coverage from 65% to 90%, with access to all functions and thus a maximum of 100%, would yield a score of 25% (= 25/100). Despite the fact that the search-based method may have explored more complex paths within existing functions, the normalised metric still heavily favours the LLM, which benefits from writing minimal tests for previously uncovered functions. Thus, even under normalisation, the comparison remains unfair. The lower the function coverage of the original testbench, the more unfair the comparison is.

8.5.2 Normalisation Based on Relative Coverage Increase

An alternative normalisation strategy considers the relative increase in coverage over the available improvement window. In this case, the increase is expressed as a proportion of the difference between the initial coverage and the maximum achievable coverage. For instance, if branch coverage starts from 65%, and increases by 5% with a theoretical maximum of 75%, the relative increase is 50% of the 10% window. If an LLM improves coverage from 65% to 90%, it achieves a relative increase of 71.4% over the 35% window, since the LLM's theoretical maximum is 100%.

This approach captures the effectiveness of each amplification method relative to its opportunity space and partially corrects for the advantage of adding tests for untested functions. It reflects how much of the available coverage gap each method succeeds in closing. However, this form of normalisation remains highly sensitive to the structure of the underlying codebase. The relative weight of different functions, such as their number of branches or lines, can distort comparisons. In Solidity smart contracts, it is common for functions to have no branches and to only perform simple sequential operations. In such cases, writing a single test may suffice to achieve full branch or line coverage for that function, artificially inflating the apparent effectiveness of LLMs.

8.5.3 Post-Processing Normalisation by Excluding Newly Covered Functions

A third possible normalisation strategy involves post-processing the coverage reports to exclude any functions that are newly covered by LLM-generated tests but remain uncovered by the search-based methods. The motivation behind this approach is to artificially equalise function coverage across both techniques, thereby allowing for a more direct comparison of their capacity to increase other metrics, such as branch or line coverage, within the same functional scope. This

could help isolate how well each method explores control flow within already-covered areas of the code.

In practice, this would involve manually parsing the coverage report JSON and removing those functions that were only covered by the LLM outputs. As a result, only the shared subset of functions would be included in the final coverage metrics. While this method does succeed in correcting for the structural advantage that LLMs have in accessing and generating tests for previously untested functions, it introduces a new form of bias. This time, against the LLM.

> The fundamental issue with this approach is that LLMs operate under more or less strict output length constraints. Given a prompt to "increase coverage as much as possible", the model will typically prioritise untested functions, as covering them offers immediate gains. However, if the evaluation later discards the output related to these functions, the LLM is effectively penalised for having made a reasonable allocation of its limited output tokens. Those tokens, once spent on generating basic tests for new functions, could otherwise have been used to construct more diversified tests within already-covered areas. Thus, by post-processing away these newly covered functions, the evaluation framework discards a significant portion of the model's contribution. This undermines the intent of its prompt and penalises its coverage-maximising behaviour.

This form of normalisation would only be justifiable if the LLM's output length were infinite, allowing the model to cover both new functions and provide meaningful exploration within existing ones. However, given current token limits, this is not the case. As such, excluding output that was generated in good faith under prompt constraints results in an unfair comparison, diminishing the LLM's effective performance. The core problem here is between fair evaluation and respecting resource limitations. It's impossible to fairly discard parts of the LLM's output without also accounting for the cost it incurred in producing them.

So, while this post-processing normalisation technique offers a theoretical path to levelling the comparison space, it does so at the expense of fairness. It forces the LLM into an evaluative framework that assumes infinite generative capacity, which is inconsistent with how it operates in real-world scenarios.

8.6 Matching Amplification Strategy to Contract Characteristics

A key insight gained through manual analysis is that no single test amplification technique universally outperforms others across all contracts. Instead, the effectiveness of a method strongly depends on the structural and semantic characteristics of the contract under test.

Search-based methods, and in particular genetic search, demonstrate a clear advantage in contracts with dense control flow and assertive conditions. Their ability to generate a large and varied set of tests over successive generations enables them to naturally uncover and exercise multiple branches, including error-handling paths introduced by require statements or conditional reverts. This aligns well with Solidity's pattern of enforcing execution correctness through runtime guards. The exploratory nature of mutation and crossover proves essential for these highly constrained, structurally rich contracts.

Conversely, LLM-based techniques excel in contracts with semantically specific constraints, such as hardcoded privilege checks, specific state-dependent behaviour, or hidden constants. Because LLMs interpret the contract code directly, they can infer these constraints and produce well-formed tests that satisfy them. This ability to reason about code semantics is especially valuable for contracts where random mutation is unlikely to guess correct values or identities, such as owner addresses or exact numeric thresholds.

The main lesson is that contract characteristics should guide the choice of amplification technique. For highly guarded, branching-heavy contracts, genetic search is better suited. For contracts involving specific semantics, domain logic, or hardcoded dependencies, LLMs are more

effective. Future work may benefit from developing hybrid strategies that dynamically choose or combine amplification approaches based on contract analysis.

Chapter 9

Future Work

9.1 Mixing LLMs

While the initial findings suggest that cross-amplification may lead to higher code coverage than self-amplification, the study is limited to a small subset of contracts and only two LLMs. Future work should extend this evaluation to a larger and more diverse set of contracts to assess the consistency of the observed trend. Additionally, experimenting with a broader range of language models, prompt strategies, and evaluation metrics such as mutation coverage or test robustness could provide deeper insights into the benefits and limitations of cross-model amplification in automated test generation.

9.2 Iterative Amplification with LLMs

An important limitation in the current setup is the reliance on GitHub Copilot Pro for test generation and amplification. Due to its interactive design, each contract and corresponding test must be manually inserted, and prompts must be copied and adjusted for each run. This manual process significantly slows down the workflow and prevents systematic iteration over multiple amplification rounds. As a result, it is not yet possible to explore whether repeated amplification passes using the same LLM potentially with adjusted prompts based on prior output could lead to further improvements in coverage. Future work should focus on automating the generation and amplification pipeline to enable multi-stage prompting strategies. This would allow more sophisticated experiments, such as iterative refinement of test suites, prompt tuning based on feedback from previous runs, and large-scale evaluation across diverse contracts. Such an automated setup is expected to not only improve efficiency but also unlock the full potential of LLM-based amplification strategies.

9.3 Mutation coverage

A full mutation coverage analysis remains a promising direction for evaluating test amplification quality. Due to tooling inefficiencies and runtime constraints, only a small-scale attempt was made in this work. This can be found below. Further automation or improved mutation frameworks for Solidity may enable this analysis at scale in the future.

The current analysis provides an initial indication of how different amplification strategies affect mutation coverage on a representative contract. While these early observations are promising, they are limited in scope. A more comprehensive evaluation is required to draw generalisable conclusions.

The tool that was used for this small analysis is SuMo [44, 43]. As part of this research, the goal is also to investigate mutation coverage. However, several practical challenges arise during this process. Although mutation testing provides informative insights, it comes with a significant

Table 9.1: Mutation testing results across test bench variants for *2018-13090.sol*

Test Bench Variant	Total Mutants	Live	Killed	Stillborn	Timed-out	Mutation Score
Base	131	95	31	5	0	24.60%
Long Prompt ChatGPT 4o	131	92	34	5	0	26.98%
Best Prompt ChatGPT 4o	131	89	37	5	0	29.37%
Best Prompt ChatGPT O3 Mini	131	87	39	5	0	30.95%
Best Prompt Claude 3.7	131	80	46	5	0	36.51%
Random Search	131	95	31	5	0	24.60%
Guided Search	131	92	34	5	0	26.98%
Genetic Search	131	86	40	5	0	31.75%

computational cost. To mitigate this limitation and still extract valuable insights from mutation testing, the analysis focuses on a single contract that best represents the overall dataset. This representative contract is selected based on its similarity to the average behaviour observed across all amplification methods. All numerical metrics from the amplification results are taken into account, and the Euclidean distance of each contract to the global mean is calculated. The contract with the smallest distance is then identified as the most representative. These metrics are sourced from an Excel file, *amplification_results.xlsx* (see Appendix A.4), which contains detailed performance results for each contract under various test amplification strategies.

For Test Bench 1, the selected contract is 2018-13090.sol, while for Test Bench 2, it is 2018-13079.sol. Mutation testing is applied comprehensively to these representative contracts, providing a computationally feasible assessment of how well different amplification techniques improve mutation coverage across typical use cases.

9.3.1 Results for testbench 1

The contract selected as representative for Test Bench 1 is 2018-13090.sol. Mutation testing is applied across all amplification strategies to evaluate whether the observed improvements in traditional coverage metrics translate into meaningful increases in fault detection capabilities. Results can be found in Table 9.1

The baseline test suite achieves a mutation score of 24.60%, killing 31 out of 131 generated mutants. Amplified test suites show varying degrees of improvement. LLM-based amplification with a long prompt (ChatGPT 4o) raises the mutation score to 26.98%, while more optimised prompting strategies increase this further: 29.37% with the best ChatGPT 4o prompt, 30.95% with the best ChatGPT o3 Mini prompt, and 36.51% with the best prompt using Claude 3.7. These results indicate that prompt engineering has a significant impact on the effectiveness of the generated tests.

Search-based amplification methods such as random and guided search yield more modest improvements, scoring respectively 24.60% and 26.98%. Genetic search does a slightly better job, obtaining 31.75%. This suggests that naive search-based strategies may be less effective in generating diverse and fault-revealing tests unless further tuned by for example evolutionary algorithms.

These findings align with the corresponding coverage metrics from the amplification results. For instance, the best prompt for Claude 3.7 not only yields the highest mutation score but also records high functional and line coverage. However, the correlation between line/branch coverage and mutation score is not always linear. Several configurations with similar line coverage exhibit different mutation scores, highlighting that higher code coverage does not necessarily imply stronger fault detection.

In summary, the mutation testing results confirm that amplified test suites generated by well-prompted LLMs particularly Claude 3.7 are more effective in detecting injected faults than those generated by search-based strategies.

Table 9.2: Mutation testing results across test bench variants for *2018-13079.sol*

Test Bench Variant	Total Mutants	Live	Killed	Stillborn	Timed-out	Mutation Score
Base	134	88	41	5	0	31.78%
Claude	134	63	66	5	0	51.16%
Genetic	134	72	57	5	0	44.19%
Hybrid (Genetic → LLM)	134	68	61	5	0	47.28%
Hybrid (LLM → Genetic)	134	60	69	5	0	53.49%

9.3.2 Results for testbench 2

The representative contract in the second testbench is 2018-13079.sol. Results can be found in Table 9.2. The baseline tests for this contract obtain a mutation score of 31.78%. This contract serves as a better foundation for mutation analysis than the selected contract of the first testbench. The reason for this is because this contract experiences a higher function coverage, compared to the representative contract used in the first test bench.

The mutation algorithm will generate mutations purely based on the contract, not on the tests. This means that if certain functions are untested, there will still be mutations in them but they will become practically unkillable, since those functions aren't tested. That's why it's useful to cover as many functions as possible, before going to the mutation analysis. Because of this, the comparison between search-based methods and LLMs will also be more fair, for the same reason discussed earlier in Section 5.5 about the problem of fairness in comparing search-based and LLM-based test amplification.

Moving on to the results, it is apparent that both genetic search, and Claude 3.7, greatly improve the mutation score when compared to the baseline metric. Genetic search is able to increase the mutation score by 12.41%, while Claude 3.7 improves this metric by 19.38%. Both approaches score really well, with Claude being the clear winner in this comparison.

Both hybrids also experience big increases in their mutation score. The first hybrid, where genetic search was applied first, followed by Claude 3.7, increases the mutation score by 15.50%, while the second hybrid, where Claude 3.7 was applied first, followed by genetic search, increases the mutation score by 21.71%. Again, this comparison has a clear winner, being the second hybrid. The general coverage increasing approach by Claude 3.7, followed by the refinements and increase in branch coverage by genetic search, proves to be the superior method once again, obtaining the highest mutation score of all experiments. This observation aligns with the coverage results obtained in earlier analysis. [44, 112, 43, 113] [2],[51],[52],[53]

So while this small-scale analysis provides some insight into the mutation coverage, future research might build on this setup to perform broader mutation testing across multiple contracts and configurations, offering a more comprehensive evaluation of test suite effectiveness.

9.4 references

Bibliography

[1] N. Alshahwan et al. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024.

[2] Prompting Guide. Mistral 7b llm. https://www.promptingguide.ai/models/mistral-7b. Accessed: Jun. 11, 2025.

[3] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[4] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[5] Prompting Guide. Phi-2. Website. Accessed: Jun. 11, 2025.

[6] Analytics Vidhya. Building and training large language models for code: A deep dive into starcoder. Blog post. Accessed: Jun. 11, 2025.

[7] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[8] Adam Hayes. Blockchain Facts: What Is It, How It Works, And How It Can Be Used. https://www.investopedia.com/terms/b/blockchain.asp, 2025. Website, accessed Jun. 11, 2025.

[9] Maher Alharby and Aad Van Moorsel. Blockchain-Based Smart Contracts: A Systematic Mapping Study. *arXiv preprint arXiv:1710.06372*, 2017.

[10] Loi Luu et al. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[11] CyberNews. Ethereum smart contract vulnerabilities can lead to millions in losses. https://cybernews.com/security/ethereum-smart-contract-vulnerabilities/, 2025. Website article, accessed Jun. 11, 2025.

[12] OWASP. OWASP Smart Contract Top 10. https://owasp.org/www-project-smart-contract-top-10/, 2025. Website, accessed Jun. 11, 2025.

[13] Xiaoqi Li et al. Smart Contracts in the Real World: A Statistical Exploration of External Data Dependencies. *arXiv preprint arXiv:2406.13253v3*, 2024.

[14] Ethereum Stack Exchange Community. Principles of Testing Smart Contracts. https://ethereum.stackexchange.com/questions/141119/principles-of-testing-smart-contracts, 2025. QA website, accessed Jun. 11, 2025.

[15] Hedera Hashgraph. Testing Smart Contracts. https://hedera.com/learning/smart-contracts/testing-smart-contracts, 2025. Website, accessed Jun. 11, 2025.

[16] Natasha. Automated Testing for Blockchain: How to Ensure Secure Smart Contracts. https://bugasura.io/blog/blockchain-testing-automation/, 2025. Website blog, accessed Jun. 11, 2025.

[17] Hamed Taherdoost. Smart contracts in blockchain technology: A critical review. *Information*, 14(2), 2023.

[18] Ki Byung Kim and Jonghyup Lee. Automated generation of test cases for smart contract security analyzers. *IEEE Access*, 8:209377–209392, 2020.

[19] Carolin Brandt and Andy Zaidman. Developer-centric test amplification. *Empirical Software Engineering*, 27(4):96, 2022.

[20] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12, 2015.

[21] Solidity Authors. Solidity v0.8.30 Documentation. https://docs.soliditylang.org/en/v0.8.30/, 2025. Website, accessed Jun. 11, 2025.

[22] Hardhat. Hardhat: Ethereum development environment for professionals. https://hardhat.org/, 2025. Website, accessed Jun. 11, 2025.

[23] mochajs. Mocha – simple, flexible, fun javascript test framework. https://mochajs.org/. Accessed Jun. 2025.

[24] chaijs. Chai assertion library. https://www.chaijs.com/. Accessed Jun. 2025.

[25] Alchemy. Solidity Code Coverage. https://www.alchemy.com/dapps/solidity-code-coverage, 2024. [Accessed: Nov. 12, 2024].

[26] Cătălina Mărcuță MoldStud Research Team. Why Every Solidity Developer Should Master Hardhat for Efficient Smart Contract Development. https://moldstud.com/articles/p-why-every-solidity-developer-should-master-hardhat-for-efficient-smart-contract-devel 2025. Website, accessed Jun. 11, 2025.

[27] GitHub. GitHub Copilot Documentation. https://docs.github.com/en/copilot, 2025. Website, accessed Jun. 11, 2025.

[28] Towards Data Science. How Copilot Helps Me Write Tests. https://towardsdatascience.com/how-copilot-helps-me-write-tests, 2025. Blog article, accessed Jun. 11, 2025.

[29] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. Automatic test improvement with dspot: a study with ten mature open-source projects. *Empirical Software Engineering*, 24(4):2603–2635, 2019.

[30] Elias Schoofs, Mahdi Abdi, and Serge Demeyer. Ampyfier: test amplification in python. *Journal of Software: Evolution and Process*, 2022.

[31] Mehrdad Abdi, Henrique Rocha, Serge Demeyer, and Alexandre Bergel. Small-amp: Test amplification in a dynamically typed language. *Empirical Software Engineering*, 27, 07 2022.

[32] GeeksforGeeks. Genetic Algorithms. https://www.geeksforgeeks.org/genetic-algorithms/, 2025. Website, accessed Jun. 11, 2025.

[33] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)*, pages 85–103, 2007.

[34] German Emir Sibay, Victor Braberman, Sebastian Uchitel, and Jeff Kramer. Synthesizing modal transition systems from triggered scenarios. *IEEE Transactions on Software Engineering*, 39(7):975–1001, 2013.

[35] Truffle Suite. Truffle Suite Documentation Archive. https://archive.trufflesuite.com/, 2024. [Accessed: Nov. 2, 2024].

[36] Kingsley Arinze. Truffle vs. Hardhat: Breaking Down the Difference Between Ethereum's Top Development Environments. https://archive.trufflesuite.com/blog/truffle-vs-hardhat-breaking-down-the-difference-between-ethereums-top-development-envir 2025. Website blog, accessed Jun. 11, 2025.

[37] Foundry. Foundry - ethereum development framework. Website. Accessed: Jun. 11, 2025.

[38] Remix IDE. Remix - ethereum ide. Website. Accessed: Jun. 11, 2025.

[39] Trail of Bits. Echidna. GitHub repository. Accessed: Jun. 11, 2025.

[40] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts, 2020.

[41] Christof Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts, 03 2021.

[42] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ting Wang, Teng Hu, Xiuzhuo Xiao, Dong Wang, Jin Huang, and Xiaosong Zhang. A large-scale empirical study on control flow identification of smart contracts. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2019.

[43] Barboni, Morena. SuMo: SOlidity MUtator. https://github.com/MorenaBarboni/SuMo-SOlidity-MUtator, 2022. GitHub repository, accessed Jun. 11, 2025.

[44] M. Barboni, A. Morichetta, and A. Polini. SuMo: A mutation testing approach and tool for the Ethereum blockchain. *Journal of Systems and Software*, 193:111445, 2022.

[45] Metana Editorial. Solidity mutation testing: Guide for beginners. https://metana.io/blog/solidity-mutation-testing/. Accessed Jun. 2025.

[46] Sefa Akca, Chao Peng, and Ajitha Rajan. Testing smart contracts: Which technique performs best? In *Proceedings of the ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '21)*, page 11, Bari, Italy, 2021. ACM.

[47] Wessel Oosterbroek. Bachelor thesis. Master's thesis, TU Delft, Electrical Engineering, Mathematics and Computer Science, 2021. Bachelor thesis, supervised by C.E. Brandt.

[48] Markos Viggiato, Dale Paas, Chris Buzon, and Cor-Paul Bezemer. Identifying similar test cases that are specified in natural language, 2021.

[49] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Software Engineering, IEEE Transactions on*, 36:226 – 247, 05 2010.

[50] Kush Jain and Claire Le Goues. Testforge: Feedback-driven, agentic test suite generation, 2025.

[51] H. Liang, W. Chen, and et al. Llm-based readability measurement for unit tests' context. *arXiv preprint arXiv:2403.00001*, 2024. arXiv:2403.00001.

[52] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1:1703–1726, 07 2024.

[53] Vaishnavi Pulavarthi, Deeksha Nandal, Soham Dan, and Debjit Pal. Are llms ready for practical adoption for assertion generation?, 2025.

[54] Vaishnavi Pulavarthi, Deeksha Nandal, Soham Dan, and Debjit Pal. Assertionbench: A benchmark to evaluate large-language models for assertion generation, 06 2024.

[55] Jingming Zhuo, Songyang Zhang, Xinyu Fang, Haodong Duan, Dahua Lin, and Kai Chen. Prosa: Assessing and understanding the prompt sensitivity of llms, 2024.

[56] Anwoy Chatterjee, H S V N S Kowndinya Renduchintala, Sumit Bhatia, and Tanmoy Chakraborty. Posix: A prompt sensitivity index for large language models, 2024.

[57] Navid Bin Hasan, Md. Ashraful Islam, Junaed Younus Khan, Sanjida Senjik, and Anindya Iqbal. Automatic high-level test case generation using large language models, 2025.

[58] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[59] Paolo Tonella. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, 29(4):119–128, July 2004.

[60] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation, 2023.

[61] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. Hits: High-coverage llm-based unit test generation via method slicing, 2024.

[62] S. Hu, Z. Lin, and J. Zhao. An empirical study on the test coverage and fault detection capabilities of llm-generated tests. *arXiv preprint arXiv:2406.18181v1*, 2024. arXiv:2406.18181v1.

[63] Christopher Foster, Abhishek Gulati, Mark Harman, Inna Harper, Ke Mao, Jillian Ritchey, Hervé Robert, and Shubho Sengupta. Mutation-guided llm-based test generation at meta. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25)*, pages 23–27, Trondheim, Norway, 2025. ACM. ACM, New York, NY, USA.

[64] Ye Shang, Quanjun Zhang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. A large-scale empirical study on fine-tuning large language models for unit testing, 2024.

[65] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. Aster: Natural and multi-language unit test generation with llms. *arXiv preprint arXiv:2409.03093v2*, 2024. arXiv:2409.03093v2.

[66] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. Multi-language unit test generation using llms. In *Proceedings of ACM Conference (Conference'17)*, page 13, New York, NY, USA, 2024. ACM.

[67] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

[68] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, pages 1–59. Springer, 2012.

[69] Wanida Khamprapai, Cheng-Fa Tsai, Paohsi Wang, and Chi-En Tsai. Performance of enhanced multiple-searching genetic algorithm for test case generation in software testing. *Mathematics*, 9(15), 2021.

[70] Saeed Zarinfam. Static analysis vs dynamic analysis. Website. Accessed: Jun. 11, 2025.

[71] S. W. Driessen, D. Di Nucci, D. A. Tamburri, and W. J. van den Heuvel. Solar: Automated test-suite generation for solidity smart contracts. *Science of Computer Programming*, 232, January 2024. Publisher Copyright: © 2023 The Author(s).

[72] M. Olsthoorn, D. Stallenberg, A. Van Deursen, and A. Panichella. SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 202–206, Pittsburgh, PA, USA, 2022.

[73] Konstantin Britikov, Ilia Zlatkin, Grigory Fedyukovich, Leo Alt, and Natasha Sharygina. *SolTG: A CHC-Based Solidity Test Case Generator*, pages 466–479. Researchgate, 07 2024.

[74] S. Driessen et al. Automated test-case generation for solidity smart contracts: The AGSolT approach and its evaluation. *arXiv preprint arXiv:2102.08864*, 2021.

[75] Mohamed Salah Bouafif, Mohammad Hamdaqa, and Edward Zulkoski. Primg : Efficient llm-driven test generation using mutant prioritization, 2025.

[76] SmartContractSecurity. Solidity test generator (proof of concept). `https://github.com/SmartContractSecurity/Solidity-Test-Generator`. GitHub repository, accessed Jun. 11, 2025.

[77] Google Scholar. Google Scholar. `https://scholar.google.be/`, 2025. Website, accessed Jun. 11, 2025.

[78] Semantic Scholar. Semantic Scholar. `https://www.semanticscholar.org/`, 2025. Website, accessed Jun. 11, 2025.

[79] Monica Jin vincenth Gernot Salzer, Joao F. Ferreira. Sb curated: A curated dataset of vulnerable solidity smart contracts. `https://github.com/smartbugs/smartbugs-curated`. GitHub repository, accessed Jun. 11, 2025.

[80] Rui Maranhao Joao F. Ferreira, Thomas Durieux. Smartbugs wild dataset. `https://github.com/smartbugs/smartbugs-wild`. GitHub repository, accessed Jun. 11, 2025.

[81] Chavhan Yashavant, Saurabh Kumar, and Amey Karkare. Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities, 02 2022.

[82] Ren Meng. Smart Contract Benchmark Suites - Dataset. https://github.com/renardbebe/Smart-Contract-Benchmark-Suites/tree/master/dataset, 2025. GitHub repository, accessed Jun. 11, 2025.

[83] Seyyed Ali Ayati. solidity-dataset. https://huggingface.co/datasets/seyyedaliayati/solidity-dataset. Dataset, accessed Jun. 11, 2025.

[84] Zhiyuan Peng, Xin Yin, Rui Qian, Peiqin Lin, Yongkang Liu, Chenhao Ying, and Yuan Luo. Soleval: Benchmarking large language models for repository-level solidity code generation, 2025.

[85] Fabiano Pecorelli, Giovanni Grano, Fabio Palomba, Harald C. Gall, and Andrea De Lucia. Toward granular search-based automatic unit test case generation. *Empirical Software Engineering*, 29(4):71, 2024.

[86] Zhichao Zhou, Yutian Tang, Yun Lin, and Jingzhu He. An llm-based readability measurement for unit tests' context-aware inputs, 2024.

[87] Dietmar Winkler, Pirmin Urbanke, and Rudolf Ramler. Investigating the readability of test code. *Empirical Software Engineering*, 29(2), February 2024.

[88] Uniswap Team. Uniswap v3 Core Contracts. https://github.com/Uniswap/v3-core/tree/main/contracts, 2025. GitHub repository, accessed Jun. 11, 2025.

[89] Synthetix Team. Synthetix Contracts. https://github.com/Synthetixio/synthetix/tree/develop/contracts, 2025. GitHub repository, accessed Jun. 11, 2025.

[90] OpenZeppelin Team. OpenZeppelin Contracts. https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts, 2025. GitHub repository, accessed Jun. 11, 2025.

[91] Meng Ren. Biography. https://renardbebe.netlify.app/, 2025. Website, accessed Jun. 11, 2025.

[92] Hugging Face. Models. https://huggingface.co/models. Accessed: Jun. 11, 2025.

[93] Hugging Face. A noob's introduction to transformers. https://huggingface.co/blog/noob_intro_transformers. Accessed: Jun. 11, 2025.

[94] Stack Overflow. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#section-coding-tools, 2023. Survey results page, accessed Jun. 11, 2025.

[95] InfoWorld. GitHub Copilot gains test generation with GPT-4. https://www.infoworld.com/article/3717716, 2024. News article, accessed Jun. 11, 2025.

[96] The Verge. OpenAI launches O3 mini in ChatGPT and API. https://www.theverge.com/2024/5/13/openai-o3-mini-launch-chatgpt-api-available-now, 2024. News article, accessed Jun. 11, 2025.

[97] Family friendly user. Real Talk: o3-mini (high effort) is a nightmare for actual coding. https://www.reddit.com/r/ChatGPT/comments/1if3pis/real_talk_o3mini_high_effort_is_a_nightmare_for, 2025. Discussion thread, accessed Jun. 11, 2025.

[98] OpenAI. GPT-4o. https://openai.com/index/gpt-4o, 2024. Website, accessed Jun. 11, 2025.

[99] Anthropic. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku. https://www.anthropic.com/news/3-5-models-and-computer-use, 2025. News post, accessed Jun. 11, 2025.

[100] Anthropic. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet, 2025. News post, accessed Jun. 11, 2025.

[101] Google Cloud. Gemini 2.0 Flash. https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash, 2025. Documentation, accessed Jun. 11, 2025.

[102] Jonathan Aldrich. Lecture notes: Concolic testing. Course notes, Carnegie Mellon University. Accessed: Jun. 11, 2025.

[103] Maria Lapina, Arsen Aganesov, Vitalii Lapin, and Vijay Athavale. *Research of Dynamic Fuzzing Methods to Identify Vulnerabilities in Program Code*, pages 172–184. Researchgate, 10 2024.

[104] Ali Muhammad. An industrial case study of the effectiveness of test generators. *Academia.edu*, 2012. Accessed: Jun. 11, 2025.

[105] Gregory Gay. Lecture 12: Automated test case generation. TDA 594/DIT 593, December 9, 2021, Dec 2021. Accessed: Jun. 11, 2025.

[106] Zelda B. Zabinsky. Random search algorithms. *University of Washington Technical Report*, Apr 2009. Accessed: Jun. 11, 2025.

[107] Educative, Inc. Common termination conditions in genetic algorithm. https://how.dev/answers/common-termination-conditions-in-genetic-algorithm, 2025. Accessed: Jun. 11, 2025.

[108] Jordan LaPrise. Perform crossover operation on ast in genetic programming. https://softwareengineering.stackexchange.com/questions/317157/perform-crossover-operation-on-ast-in-genetic-programming. Accessed: Jun. 11, 2025.

[109] OpenAI (2023). Gpt-4 technical report. Technical report, OpenAI, 2023. Accessed: Jun. 11, 2025.

[110] Rich Lysakowski. How to change github copilot settings in vscode to increase the token limit to 4098 or 8196 (as promised by microsoft)? https://stackoverflow.com/questions/77842786/how-to-change-github-copilot-settings-in-vscode-to-increase-the-token-limit-to-4. Accessed Jun. 2025.

[111] sytafe. Frequent rate limit issues when calling github copilot via vscode + cline + vscodelm. https://github.com/orgs/community/discussions/150373. Accessed Jun. 2025.

[112] Pasha Finkelshteyn. A Comprehensive Guide to Mutation Testing in Java. https://bell-sw.com/blog/a-comprehensive-guide-to-mutation-testing-in-java/#:~:text=Mutation%20testing%20helps%20assess%20the,spots%20not%20covered%20by%20tests, 2024. Blog article, accessed Jun. 11, 2025.

[113] Morena Barboni, Andrea Morichetta, and Andrea Polini. Sumo: A mutation testing strategy for solidity smart contracts. *CoRR*, abs/2105.03626, 2021.

Appendix A

Supplementary Materials

A.1 Replication Package Repository

A complete replication package for this study is publicly available on GitHub:
https://github.com/jordvhan/Solidity_smart_contracts_amplifier_SB_vs_LLM
This repository includes:

- Full source code and implementation of all test amplification techniques.
- All scripts, configuration files, and generated test cases used in the experiments.
- A detailed `README` explaining the setup, folder structure, and how to run the experiments.
- Evaluation scripts and data used to compute key metrics (e.g., coverage, generation time, test inflation).

This package ensures reproducibility of the experiments and facilitates further research based on this work.

A.2 Additional Coverage Scatter Plots for Dataset 1

To supplement 4.6 in the main text, this appendix includes scatter plots for additional coverage metrics plotted against lines of code (Figure A.2.1. These include:

- Function coverage vs. lines of code
- Line coverage vs. lines of code

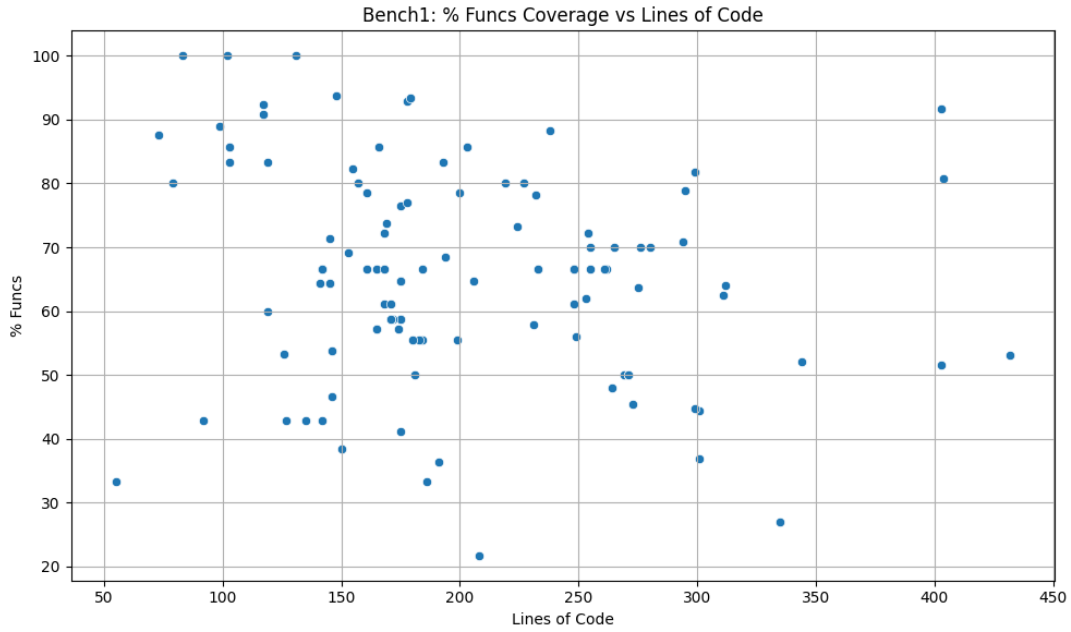
As with the main figures, the data points show a wide dispersion without clear correlation, supporting the conclusion that coverage is not linearly dependent on contract size. These plots further confirm the structural diversity of the test suite and the independence of coverage outcomes from test file length.

A.3 Additional Coverage Scatter Plots for Dataset 2

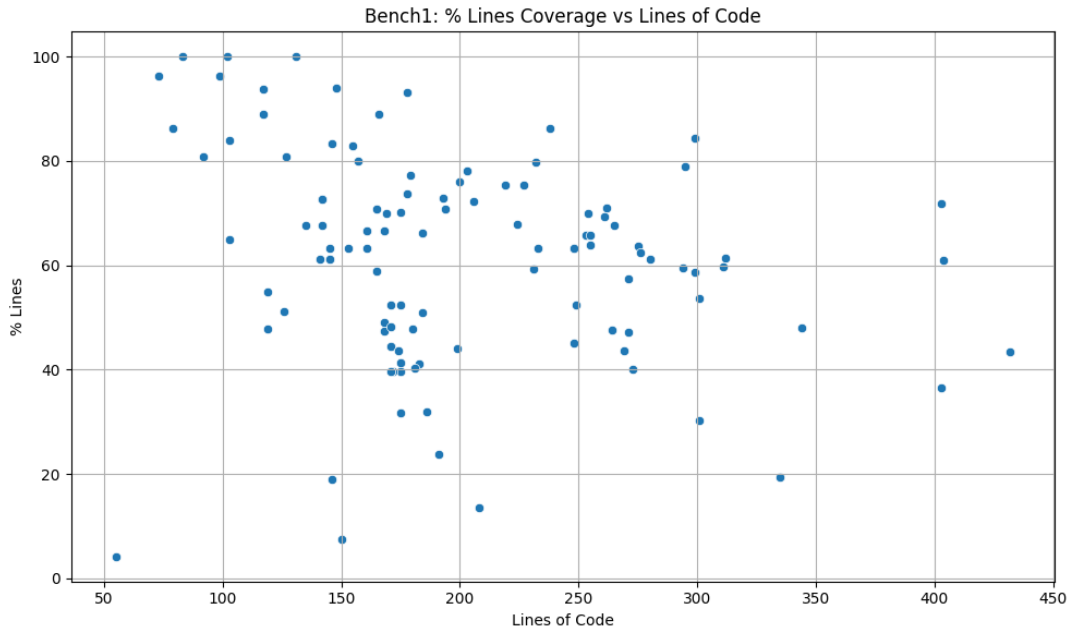
In addition to Figure 4.8 in the main text, this appendix includes scatter plots for the remaining coverage metrics plotted against lines of code for the second dataset (Figure A.3.1. These include:

- Function coverage vs. lines of code
- Line coverage vs. lines of code

As with the earlier dataset, the scatter plots show no clear correlation between contract size and coverage values. This further supports the claim that the dataset is structurally balanced and suitable for unbiased analysis of test performance.

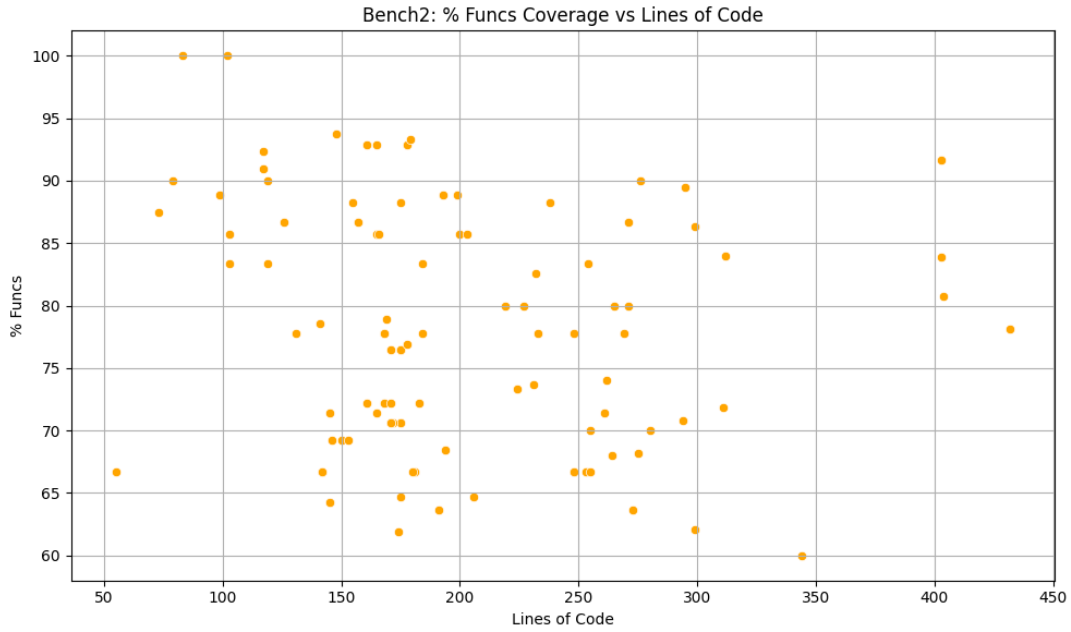


(a) function coverage to LOC

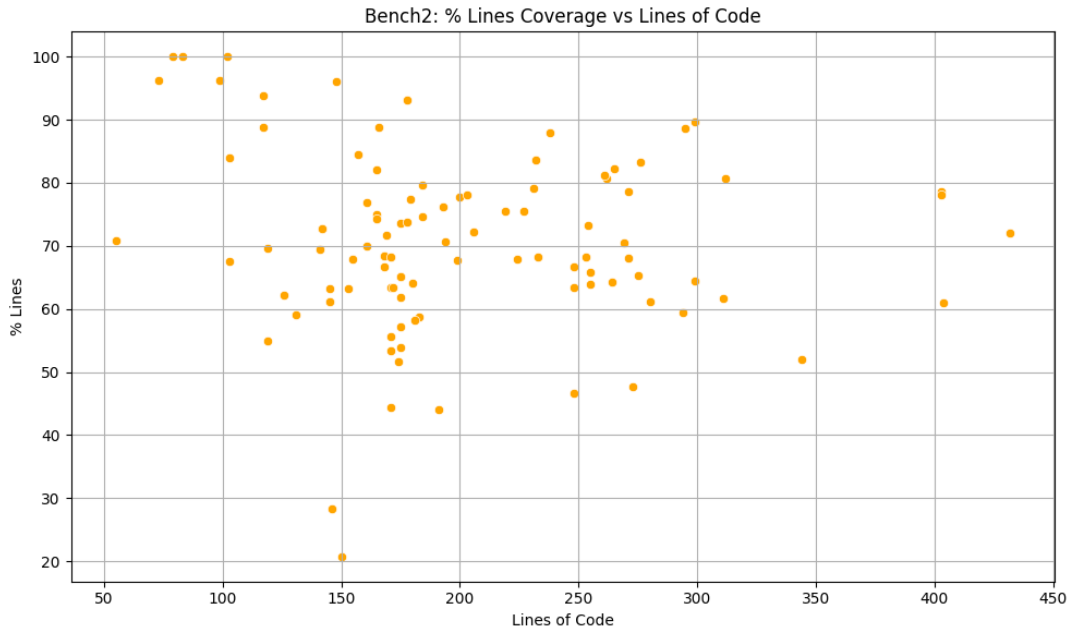


(b) Line coverage to LOC

Figure A.2.1: Remaining coverage metrics plotted to the Lines Of Code (LOC)



(a) function coverage to LOC



(b) Line coverage to LOC

Figure A.3.1: Remaining coverage metrics plotted to the Lines Of Code (LOC) for the new dataset

A.4 Amplification Results Used for Analysis

The selection of a representative contract for mutation testing is based on numerical metrics derived from amplification results. These results are stored in the file `amplification_results.xlsx`, available in the GitHub repository[1].

The file contains the statement, branch, function, and line coverage values, per contract, per amplification method, along with a summary of the average values per metric per method. From this, a Euclidean distance was computed between each contract and the global average across all metrics. The contract with the smallest distance was selected as the most representative.

A simplified excerpt of the table is shown in Figure A.4.1

LLM Amplification				Search Based Amplification			
Amplification Claude 3.7				Amplification Genetic Search			
% Stmts	% Branche	% Funcs	% Lines	%Stmts	%Branches	%Funcs	%Lines
85,37	82,5	86,36	89,66	85,37	77,5	86,36	89,66
72,58	58,62	87,5	73,42	58,06	41,38	70,83	59,49
79,59	71,43	92,31	67,92	34,69	28,57	69,23	28,3
77,59	57,81	80,95	82,93	63,79	48,44	66,67	68,29
77,42	72,22	86,67	84,44	77,42	66,67	86,67	84,44
64,86	58,33	88,46	73,98	50	41,67	80,77	61,79

Figure A.4.1: Excerpt from `amplification_results.xlsx` showing metrics used in the analysis.

[1] https://github.com/jordvhan/Solidity_smart_contracts_amplifier_SB_vs_LLM/tree/main/excel_code