University of Camerino

School of Advanced Studies

Doctor of Philosophy in Computer Science and Mathematics
- XXXVII Cycle

Enhancing Smart Contract Reliability: Dynamic Approaches for Quality Assurance

Supervisor

Prof. Andrea Polini

PhD Candidate

Morena Barboni

Co-Supervisor

Prof. Andrea Morichetta

Academic Year 2024-2025

ABSTRACT OF THE DISSERTATION

Blockchain technologies had a significant impact on many sectors of contemporary society, with virtual currencies being the most prominent example. The introduction of the Ethereum blockchain and its native support for smart contracts, self-enforcing programs that enable trustworthy digital interactions, has broadened the possible adoption contexts. These programs possess unique characteristics, such as code immutability and autonomous execution, which necessitate innovative testing methodologies. Despite their growing adoption, current testing practices and tools for smart contracts lag behind those available for traditional software systems, raising concerns about the reliability of decentralized applications. This dissertation addresses these challenges through three core research objectives. First, it advances *mutation testing* for Ethereum smart contracts by introducing a practical framework and tool that support Solidity-specific test adequacy assessment and incremental mutation analysis during development. This enables developers to systematically evaluate and improve the fault-detection capabilities of their test suites based on metrics that go beyond simple code coverage. Second, it explores how *smart contract auditing* practices can benefit from mutation testing by integrating live mutant inspection into code reviews and automating the generation of missing test cases. This contribution enhances the auditors' ability to identify weaknesses in the test suite and provide actionable feedback to clients. Third, it supports smart contract *maintenance* activities by proposing a novel capture-replay testing framework and tool for upgradeable contracts. The approach harnesses historical blockchain transactions as tests, allowing developers to detect behavioral inconsistencies introduced by upgrades without the need to manually reconstruct testing scenarios. Through these contributions, the thesis aims to advance the state of smart contract quality assurance, offering both practical frameworks and theoretical insights that enhance the reliability of blockchain-based systems.

LIST OF PUBLICATIONS

Journal Articles

1. **Morena Barboni**, Guglielmo De Angelis, Andrea Morichetta and Andrea Polini. *"CATANA: Wielding Historical Transactions for Replay Testing of Upgradeable Smart Contracts"*. ACM Transactions on Internet Technology (ToIT), 2025

2. **Morena Barboni**, Andrea Morichetta, Andrea Polini and Francesco Casoni. *"ReSuMo: a regression strategy and tool for mutation testing of solidity smart contracts."* Software Quality Journal (SQJ), 2023

3. **Morena Barboni**, Antonia Bertolino and Guglielmo De Angelis. *"Flakiness goes live: Insights from an In Vivo testing simulation study"*. Information and Software Technology (IST), 2023

4. **Morena Barboni**, Andrea Morichetta and Andrea Polini. *"SuMo: A mutation testing approach and tool for the Ethereum blockchain"*. Journal of Systems and Software (JSS), 2022

Conference Proceedings

1. **Morena Barboni**, Filippo Lampa, Andrea Morichetta, Andrea Polini and Edward Zulkoski. *"Mutant-Driven Test Generation for Ethereum Smart Contracts via LLMs"*. IEEE International Conference on Artificial Intelligence Testing (AITest), 2025

2. **Morena Barboni**, Filippo Lampa, Andrea Morichetta, Andrea Polini and Edward Zulkoski. *"Alchemist: LLM-Driven Test Generation using Solidity Mutants and the Scientific Method"*. Proceedings of the 2025 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), 2025

3. **Morena Barboni**, Andrea Morichetta, Andrea Polini, Sebastian Banescu and Edward Zulkoski. *"Mutation Testing of Smart Contracts As a Service"*. Proceedings of the 17th International Conference on the Quality of Information and Communications Technology (QUATIC), 2024

4. Sebastian Banescu, **Morena Barboni**, Andrea Morichetta, Andrea Polini and Edward Zulkoski. *"Enhanced mutation testing of smart contracts in support of code inspection"*. Proceedings of the 2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), 2024

5. **Morena Barboni**, Guglielmo De Angelis, Andrea Morichetta and Andrea Polini. *"CATANA: Replay Testing for the Ethereum Blockchain"*. Proceedings of the 35th IFIP International Conference on Testing Software and Systems (ICTSS), 2023

6. **Morena Barboni**, Francesco Casoni, Andrea Morichetta and Andrea Polini. *"ReSuMo: Regression Mutation Testing for Solidity Smart Contracts"*. Proceedings of the 15th International Conference on the Quality of Information and Communications Technology (QUATIC), 2022

7. **Morena Barboni**, Andrea Morichetta and Andrea Polini. *"Smart Contract Testing: Challenges and Opportunities"*. Proceedings of the 5th IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2022

8. **Morena Barboni**, Antonia Bertolino and Guglielmo De Angelis. *"What we talk about when we talk about software test flakiness"*. Proceedings of the 14th International Conference on the Quality of Information and Communications Technology (QUATIC), 2021

9. **Morena Barboni**, Andrea Morichetta and Andrea Polini. *"SuMo: A Mutation Testing Strategy for Solidity Smart Contracts"*. Proceedings of the IEEE/ACM International Conference on Automation of Software Test (AST@ICSE), 2021

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ACRONYMS

AAC: Address Aliasing Check . . . 178
ACM: Argument Change of overloaded Method call . . . 95
AMR: Array Member Replacement . . . 177
AOR: Assignment Operator Replacement . . . 81
AST: Abstract Syntax Tree . . . 45
AVR: Address Value Replacement . . . 78
BCRD: Break and Continue Replacement and Deletion . . . 80
BLR: Boolean Literal Replacement . . . 77
BOR: Binary Operator Replacement . . . 81
BVR: Boolean Value Replacement . . . 177
CA: Contract Account . . . 12
CBD: Catch Block Deletion . . . 81
CCD: Contract Constructor Deletion . . . 94
CER: Casting Expression Replacement . . . 178
CFG: Control Flow Graph . . . 43
CSC: Conditional Statement Change . . . 80
dApps: Decentralized Applications . . . 3
DCOE: Dockerized Combined Operators Executor . . . 193
DeFi: Decentralized Finance . . . 12
DLR: Data Location keyword Replacement . . . 80
DCME: Dockerized Combined Mutants Executor . . . 193
DOD: Delete Operator Deletion . . . 81
EAS: Event Argument Swap . . . 171
ECG: Extended Call Graph . . . 43
ECS: Explicit Conversion to Smaller type . . . 79
EED: Event Emission Deletion . . . 89
EHC: Exception Handling statement Change . . . 90
EHD: Exception Handling Deletion . . . 180
EOA: Externally Owned Account . . . 12
ER: Enum Replacement . . . 78

PART I

CONTEXT AND CHALLENGES

CHAPTER 1

INTRODUCTION

In recent years, both academics and industry practitioners have shown growing interest in blockchain technologies. The unique features of this new environment, such as decentralization and code immutability, offer new commercial prospects and benefits for enterprises. At the same time, these very characteristics introduce unique challenges for the verification and validation of blockchain-based software systems.

This thesis focuses on Ethereum [191], one of the most mature and widely adopted public blockchain platforms. Introduced in 2013, Ethereum is known for its native support to *smart contracts*, self-executing programs that enable the development of complex Decentralized Applications (dApps). Given the unique nature of blockchain and the high financial stakes involved, ensuring the reliability of smart contract code is essential.

The remainder of this introduction provides the context for the thesis. Section 1.1 discusses the motivation behind this research, while Section 1.2 outlines the high-level Research Objectives (ROs). Finally, Section 1.3 describes the structure of the thesis.

1.1 Motivation

Blockchain technology has evolved rapidly over the past decade, finding applications in various domains such as finance, supply chain management, and decentralized governance. In Ethereum, smart contracts form the backbone of dApps, as they define the business logic that governs transactions and enforces agreements. These self-executing programs ensure that interactions between Ethereum accounts comply strictly with the encoded terms, a principle known as *code is law*.

Given their immutability and business-critical role, smart contracts demand higher reliability guarantees than traditional systems. This has led to increasing concern within the software testing community [51] and among blockchain industry professionals [207], for whom the safe and reliable deployment of smart contracts represents the most pressing challenge. The need for rigorous testing in this domain is driven by several critical factors:

1) *High cost of failure:* Failures in smart contract execution can lead to severe financial and reputational damage. Beyond unintentional bugs, attackers often exploit subtle

vulnerabilities to manipulate contract behavior and steal assets. A notable example is the DAO attack [113], where a reentrancy flaw allowed an attacker to drain 3.6 million Ether (approximately $50 million at the time). In 2023 alone, smart contract exploits resulted in losses exceeding $437.5 million [145], often due to common issues like rounding errors, unchecked assumptions, or inadequate input validation.

2) *Irreversibility of transactions:* Once a blockchain transaction is confirmed, its effects are permanently recorded on the ledger. This means that any losses resulting from a software failure are typically unrecoverable, making proactive quality assurance essential.

3) *Code immutability:* Unlike traditional applications that can be patched post-deployment, smart contracts are immutable. While upgrade mechanisms do exist, they often require deploying new contract instances or navigating governance procedures with time delays [116]. In the meantime, vulnerable or faulty contracts may remain operational and exposed to exploitation.

4) *Complex and fragmented software stack:* Ethereum developers work within a diverse and heterogeneous ecosystem of languages, frameworks, and tools. This fragmented stack, coupled with frequent updates and evolving best practices, makes writing secure, maintainable code a continuous challenge. Interviews with practitioners [207] confirm that maintaining high-quality standards in such an environment is far from trivial.

These factors highlight the importance of rigorous quality assurance for smart contract-based applications. Yet, the current landscape of testing methodologies and tools for smart contract development remains considerably less mature than that of traditional software systems [207]. Many existing tools suffer from limited scope, lack of automation, or insufficient integration with real-world development workflows.

Furthermore, fundamental activities such as test adequacy assessment, regression testing, and upgrade impact evaluation are either poorly supported or entirely absent from most frameworks [20], leaving developers without the means to systematically validate contract behavior across the software life-cycle.

The consequences of insufficient quality assurance extend beyond developers. End users, who entrust contracts with financial assets, may suffer irreversible losses due to subtle bugs or vulnerabilities [186]. At the ecosystem level, repeated failures and high-profile exploits erode confidence in the blockchain as trustworthy, dependable technology.

To address this challenge, this thesis explores key aspects of *smart contract quality assurance*. It investigates how activities related to *testing*, *auditing*, and *maintenance* can be enhanced to improve the overall reliability of Ethereum-based applications. The next section presents the three high-level research objectives that guide and structure this thesis.

1.2 Research Objectives

This thesis is structured around the following high-level ROs, each addressing a specific aspect of smart contract quality assurance across the software lifecycle.

RO1: Supporting Test Adequacy Assessment for Smart Contracts

The first RO focuses on strengthening smart contracts testing practices. Testing serves as the first and most critical line of defense against the release of faulty smart contract code. However, the presence of tests alone is not sufficient. Without evaluating the *adequacy of* these *tests*, developers risk relying on test suites built on shaky ground. Yet, the current Ethereum development landscape lacks advanced frameworks for assessing test adequacy.

Solidity test suites are typically evaluated using structural coverage metrics, such as statement and branch coverage, which serve as the de facto standard in smart contract development [188, 27]. Even in comparative studies that explore advanced testing techniques, coverage remains the dominant metric used to assess tool performance [1]. This reliance on coverage is further reinforced by the widespread use of tools such as `solidity-coverage`, which are commonly integrated into developer workflows. While coverage is convenient and accessible, it is universally considered a weak proxy for test quality [81, 182], and can thus give developers a false sense of security.

This thesis introduces and investigates the use of *mutation testing* as a more rigorous approach to test adequacy assessment. Mutation testing works by injecting artificial faults (mutants) into the smart contract code to simulate common bugs, then evaluating whether the test suite can detect them [64]. Mutants that survive the process (i.e., *live* mutants) can provide useful tips for test improvement, and for achieving greater fault detection. Empirical studies have shown that the ability to kill these mutants is strongly correlated with the likelihood of detecting real faults [88], making mutation testing especially relevant in high-stakes domains such as blockchain.

Although mutation testing has been extensively studied in traditional systems, its application to smart contracts remains limited. Existing tools designed for Ethereum often lack standardization, efficiency, and integration with real-world development environments [20]. Some tools inject only a narrow set of mutant types and do not account for specific Solidity constructs [3, 40, 100, 75]. These limitations create a fragmented and immature ecosystem, making it difficult for developers to adopt mutation testing in practice.

Furthermore, existing solutions typically treat mutation testing as a standalone process, which is best suited for pre-deployment evaluation. This traditional model discourages continuous adequacy assessment in day-to-day development, as re-running full mutation testing after every change is often infeasible. Mutation testing becomes especially impractical in the context of upgradeable contracts, where each release ideally requires thorough re-validation, but re-executing the full process can delay important updates and fixes. These constraints might lead developers to skip mutation testing entirely, compromising the reliability of deployed smart contracts.

To address these challenges, RO1 investigates how mutation testing can be adapted to the specific constraints of smart contract development. Additionally, it explores more lightweight and incremental forms of mutation testing, which are essential for enabling continuous adequacy assessment throughout the contract life-cycle. Improving the way developers evaluate the strength of their test suites can support earlier fault detection and more reliable contract deployments.

RO2: Supporting Mutation Testing in Smart Contract Auditing

The second RO focuses on supporting the adoption of mutation testing in the context of smart contract auditing. Given the high financial stakes and the potential irreversibility of contract failures, developers often rely on specialized *auditing* firms (e.g., *Quantstamp*, *Hacken* and *Certik)* to conduct a thorough review of their code before deployment [130]. These audits are a fundamental part of the smart contract development life cycle, aiming to identify security vulnerabilities and build confidence in the correctness and security of smart contract applications.

Auditors typically employ a combination of methods (e.g., automated analysis tools, testing, and manual code inspection) to systematically evaluate smart contracts for their clients [39, 49]. As part of this process, they often assess test coverage to identify under-tested portions of the code and offer general recommendations for improving tests [2]. However, while coverage can reveal where tests are missing, it does not indicate how effective the existing tests are at detecting faults.

Mutation testing has potential to address this gap, but it is rarely used in auditing practices. This is largely due to its perceived high computational cost and limited direct benefit to the auditor. Unlike developers, auditors are usually not involved in maintaining or improving the client's test suite. As a result, techniques that require substantial effort without yielding immediately actionable insights are not too appealing.

To bridge this disconnect, RO2 explores how mutation testing can be integrated into auditing processes to better align with the auditor's role and constraints. In particular, the thesis explores the benefits of surfacing live mutants, faults not detected by the current test suite, during manual code review [139]. These live mutants may provide auditors with additional evidence about weakly tested parts of the contract, enhancing their knowledge and understanding of the code.

The second direction explores the automatic generation of test cases from these live mutants. This approach aims to produce explainable test cases that auditors can include in their final report. By doing so, auditors can provide low-effort, high-impact test suggestions to clients without being directly involved in test design. By combining these strategies, RO2 aim to make mutation testing a more practical and valuable tool within the smart contract auditing process.

RO3: Supporting Smart Contract Maintenance Practices

The third RO focuses on improving quality assurance in smart contract maintenance workflows. While smart contracts are immutable by default once deployed, the increasing adoption of *upgradeability mechanisms* (such as the proxy pattern) has made it possible to evolve contract logic in a controlled way [149]. These upgrade patterns enable contracts to preserve their deployed address and state while delegating execution to a newly deployed logic contract.

Developers commonly perform upgrades to fix bugs, enhance usability, or introduce new functionality [101]. However, these upgrades must be performed carefully, as they carry significant risks. Even small changes can introduce breaking behaviors or subtle regressions that impact the contract's core functionality, its integrations with other smart contracts, or front-end systems [99, 142].

Capture-replay testing is a technique in which real user interactions with a software system are recorded during execution and later replayed as test inputs. This approach allows developers to verify that newer versions of the system behave consistently with previous ones. It has been widely adopted in traditional software domains, such as web [5, 102], mobile [69], and IoT applications [58], where it serves as an effective strategy for regression testing.

Ethereum presents a unique and underutilized opportunity to bring capture-replay testing to blockchain systems. Unlike traditional software, where user interaction logs must be explicitly collected, all transactions between users and smart contracts are permanently recorded on-chain. This historical transaction data can be viewed as a natural, trustworthy log of real-world interactions. Transactions can be repurposed into executable test cases, providing a practical foundation for regression testing in upgrade scenarios.

Despite this potential, existing smart contract testing tools and methodologies remain inadequate, lacking systematic frameworks to validate changes against historical contract behavior. In practice, developers are left without structured means to determine whether behavioral changes have occurred, or whether regressions have been introduced during the upgrade process.

This final RO investigates how historical blockchain transactions can be harnessed to validate upgrades and reveal potentially disruptive smart contract changes. The goal is to support developers in performing more reliable and automated pre-upgrade verification, improving the reliability of proxy-based smart contract applications.

1.3 Thesis Structure

The thesis is organized into *five parts* and *twelve chapters*, structured as follows:

Part I — Context and Challenges

This part lays the foundational concepts and provides the necessary context for the thesis. It also expands on the key challenges it addresses and the contributions it provides.

- **Chapter 2** - Introduces the core concepts of blockchain technology, followed by a more in-depth presentation of Ethereum. It also presents the key characteristics of smart contracts and existing upgradeability mechanisms.

- **Chapter 3** - Provides an overview of quality assurance concepts in software development, focusing on those testing techniques that are most relevant to the thesis.

- **Chapter 4** - Presents the unique characteristics of the smart contract development life-cycle, highlighting quality assurance challenges that arise during the testing, auditing, and maintenance phases.

- **Chapter 5** - Provides an overview of the key research contributions of this thesis.

Part II — Mutation Testing for Smart Contracts

This part addresses RO1 by supporting test adequacy assessment in smart contract development.

- **Chapter 6** - Addresses the lack of advanced test adequacy assessment solutions in the Ethereum landscape by introducing SuMo, a mutation testing framework for the Solidity language. SuMo integrates with existing smart contract testing environments and introduces a comprehensive set of mutation operators that simulate Solidity-specific faults. SuMo helps developers to systematically evaluate the fault-detection capabilities of their test suites, identifying gaps beyond what structural metrics like code coverage can reveal.
- **Chapter 7** - Presents ReSuMo, an evolution of SuMo that implements a regression mutation testing approach. Mutation testing is computationally expensive, making frequent test adequacy assessments impractical. ReSuMo addresses this limitation by reusing prior test-mutant execution outcomes and selectively updating mutation testing results as a project evolves. In this way, it supports frequent adequacy evaluation throughout the smart contract development life-cycle.

Part III — Mutation Testing for Smart Contract Auditing

This part delves into RO2, exploring how mutation testing can enhance auditing workflows and code review practices. In particular, this research was conducted in collaboration with *Quantstamp*, a blockchain security company that specializes in smart contract auditing.

- **Chapter 8** - Investigates the role of mutation testing in smart contract auditing, exploring whether live mutants can assist auditors in reviewing contract implementations. While coverage analysis is commonly used to guide code reviews, it can only offer limited insights. Mutation analysis might further aid auditors in reasoning about smart contract correctness. Additionally, this Chapter informs refinements to the SuMo mutation operators, optimizing their effectiveness in auditing contexts.
- **Chapter 9** - While mutation testing is a powerful technique for assessing test adequacy, its adoption in auditing remains limited due to high computational costs and lack of standardized tooling. This Chapter introduces a remote service that distributes mutation testing tasks across parallel Docker containers, reducing execution time. Additionally, it presents infrastructure improvements, including enhanced result visualization, to facilitate integration into auditing workflows. The Chapter also evaluates different workload distribution strategies across Solidity projects, providing insights into optimizing mutation testing performance for large-scale audits.
- **Chapter 10** - Introduces Alchemist, a novel framework that leverages Large Language Models (LLMs) to generate test cases for live Solidity mutants. While mutation testing is a powerful technique for assessing test adequacy, analyzing surviving mutants remains a labor-intensive process, limiting its adoption in time-sensitive auditing workflows. Alchemist addresses this challenge by embedding the scientific method into the

test generation process, systematically refining test cases through iterative hypothesis generation, experimentation, and observation. This structured approach enhances test generation capabilities while making the process more explainable and transparent.

Part IV — Transaction Replay for Smart Contract Maintenance

This part addresses RO3, focusing on the challenges of validating smart contract upgrades.

- **Chapter 11** - Introduces Catana, a capture-replay testing framework designed for regression testing of Proxy-based Upgradeable Smart Contracts (USCs). While upgradeability mechanisms enable contract evolution without data migration, they also introduce risks of breaking changes and unintended regressions. Catana addresses harnesses historical blockchain transactions to build replay test suites, allowing developers to assess the impact of upgrades. Unlike traditional approaches, Catana enhances observability by analyzing both transaction outputs and contract state changes, offering deeper insights into upgrade effects.

Part V — Conclusions

This final part recaps the key insights and contributions presented throughout the thesis.

- **Chapter 12** - Provides a summary of the work, discussing on how each research objective has been addressed and highlighting the broader impact of the proposed solutions on smart contract development.

CHAPTER 2

BLOCKCHAIN AND SMART CONTRACTS

This part introduces key background knowledge about blockchain concepts referenced throughout the thesis. In particular, Sections 2.1 and 2.2 provide a first description of blockchain technology, followed by a more in-depth presentation of the Ethereum blockchain. Section 2.3 introduces Ethereum smart contracts, including their characteristics, the storage model, and available upgradeability mechanisms.

2.1 Blockchain

Blockchain technology has emerged as a revolutionary innovation, altering the way digital transactions are conducted across a range of industries. At its core, blockchain operates as a *decentralized* and *distributed* ledger, where every node or participant in the network retains a complete copy of the ledger itself. This architecture eliminates reliance on a single central authority, reducing the risks and inefficiencies often associated with the intervention of third-party intermediaries. The secure nature of blockchain comes from its use of consensus mechanisms, such as *Proof of Work* or *Proof of Stake*, which require nodes to agree on the validity of transactions before new blocks are added to the chain. This consensus process ensures that all participants trust the accuracy of the recorded data, even without knowing one another. Once data is stored on the blockchain, it becomes virtually immutable. Any change to a block's data would invalidate its cryptographic hash, altering the overall integrity of the ledger.

Although this thesis focuses primarily on *Ethereum*, there are numerous blockchain platforms available, each designed with different objectives and governance models in mind. Some networks, like *Bitcoin*, concentrate on peer-to-peer digital currency and prioritizing security through a proof-of-work mechanism. Others, such as *Hyperledger Fabric* or *R3 Corda*, opt for permissioned or private blockchain solutions, allowing only a restricted set of participants to validate transactions and maintain the ledger. This design is often preferred by enterprises requiring data privacy and controlled access. Hybrid or

consortium blockchains also exist to provide a middle ground, where multiple organizations jointly manage the network while maintaining certain decentralized features.

Regardless, the impact of blockchain is already being felt in many sectors. In finance, it can streamline payment and settlement processes. In supply chain management, it enhances visibility by tracking the provenance and movement of goods, mitigating counterfeiting and ensuring product authenticity [179]. Meanwhile, governments and public institutions are exploring blockchain-based voting systems and identity verification applications.

2.2 The Ethereum Blockchain

Among the various blockchain platforms that have been developed, *Ethereum* [35] stands out as one of the most prominent and mature technologies. Launched in 2015, Ethereum extends the capabilities of blockchain beyond simple transactions, introducing a platform that supports the deployment of smart contracts and the development of complex dApps. Unlike the Bitcoin blockchain, which is primarily designed for peer-to-peer transactions, Ethereum is a more versatile platform that allows developers to create a wide range of applications. Ethereum's native cryptocurrency, Ether (ETH), fuels the network and is used to pay for transaction fees and computational services. The Ethereum Virtual Machine (EVM) is the runtime environment for smart contracts on Ethereum, providing a secure and isolated execution environment. This flexibility has made Ethereum the foundation for Decentralized Finance (DeFi) platforms, Non-Fungible Tokens (NFTs), and various other blockchain-based solutions.

2.2.1 Ethereum Accounts

The Ethereum platform is based on an account model, where interacting parties are represented by the account they use. Each transaction can be modeled as an interaction between two different types of accounts:

1) *Externally Owned Account (EOA):* these are user-controlled gateways to the Ethereum network, managed through private keys. They are used to send, receive, and store ETH and interact with smart contracts via *transactions*. Each EOA includes a public address for identification, a private key for signing transactions, and an ETH balance.

2) *Contract Account (CA):* these represent *Smart Contracts* deployed to the Ethereum network, and function as autonomous executors of decentralized logic. CAs do not have private keys and cannot initiate transactions on their own. They can only respond to transactions from EOAs or other CAs. Each smart contract has a unique public address, an ETH balance, bytecode and storage. The *bytecode* is stored on the blockchain and executed by the EVM upon invocation, enabling CAs to perform predefined functions. The *storage*, organized as a Key-Value map, holds the contract's global values and is accessed or modified by the EVM during execution.

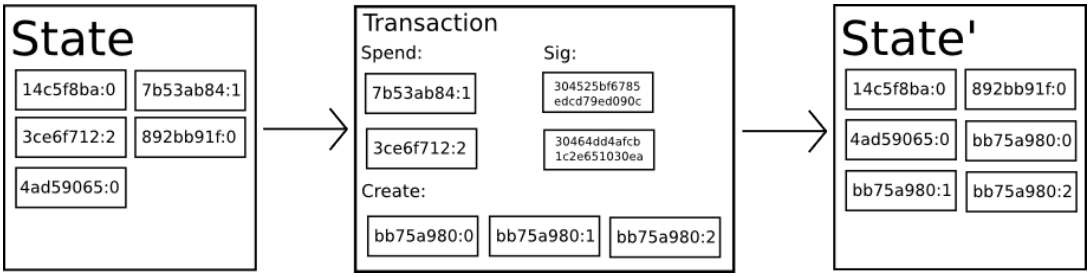


Figure 2.1: Ethereum State Transition Function

2.2.2 Transactions

Ethereum transactions are the fundamental operations that drive interactions on the blockchain, involving various actions such as the transfer of ETH and the execution of smart contract code. Transactions can be categorized into three main types.

1) A *Regular Transaction* involves an EOA sending ETH to another EOA;

2) A *Contract Creation* allows an EOA to deploy a new smart contract on the blockchain, resulting in the creation of a CA with its unique address.

3) A *Contract Execution* allows an EOA to invoke and execute a method of a deployed smart contract. This process involves running the bytecode of the smart contract on the EVM, and may include reading from or writing to the contract's *storage*, updating the *world state* as necessary.

2.2.3 The Ethereum World State

The Ethereum Yellow Paper [191] defines Ethereum as a *transaction-based state machine*. Every time transactions, contract executions, and mining activities take place, the global state of the blockchain is updated. The Ethereum world state is a comprehensive database that tracks all accounts and their states, including balances, storage, and code. Thus, the Ethereum blockchain can be seen as a state transition system, where a state transition takes a state and a transaction and outputs a new state as a result, as exemplified in Figure 2.1. In particular, a transaction can modify the state of an EOA by adjusting its ETH balance, or altering the *storage* and code associated with a CA. The EVM executes the transaction's instructions and ensures that all changes are consistent with the rules defined by the smart contract bytecode and the overall Ethereum protocol. The resulting changes are recorded in the world state and committed to the blockchain as part of a new block, providing a *historical record* of all state changes.

2.3 Ethereum Smart Contracts

Smart contracts were first conceptualized by Nick Szabo in 1994, who defined them as:

> '*A set of promises, including protocols within which the parties perform on the other promises. The protocols are usually implemented with programs*

on a computer network, or in other forms of digital electronics; thus these contracts are smarter than their paper-based ancestors'

Smart contracts are self-executing programs that operates on a blockchain network. These contracts automatically enforce the terms and conditions encoded within them, executing transactions without the need for a central authority. *Ethereum* is specifically designed to support the creation and execution of smart contracts written in a Turing-complete language. Thus, Ethereum-based applications can implement complex business logic to carry out transactions that go beyond a simple monetary transfer. At a high level, smart contracts are composed of two key components: *code* and *storage*. The code defines the contract's behavior and logic, and is written in high-level programming languages like Solidity or Vyper. This logic specifies the rules under which the contract operates and interacts with the blockchain and external parties.

Instead, the storage holds the contract's persistent data on the blockchain. Storage is used to maintain the state of the contract across transactions, and it includes the value of all its state variables. In the following more details about the characteristics of smart contract and how they operate are provided.

2.3.1 Defining Features of Smart Contracts

Smart contracts possess several defining characteristics that distinguish them from traditional software systems. These features collectively define their unique value, but also introduce unique challenges for software engineers.

Immutability The most distinctive characteristic of smart contracts is their immutability. Once a smart contract is deployed to the blockchain, its code and terms become permanently embedded in the distributed ledger and cannot be altered. This immutability ensures that the contract remains tamper-proof, fostering trust between parties. However, any bugs or vulnerabilities in the contract code cannot be directly corrected after deployment, which can lead to significant financial losses or security breaches. To mitigate this risk, developers must engage in extensive testing and validation before release. Furthermore, the inability to directly upgrade or modify the contract means that developers often design with flexibility in mind, employing techniques such as proxy contracts to enable indirect updates while preserving the original code's immutability.

Transparency Every transaction and operation executed by a smart contract is recorded on the blockchain, creating an open and auditable record accessible to all network participants. Moreover, the contract's code itself is publicly accessible. This transparency ensures that all parties can verify the contract's behavior and outcomes. However, it also exposes the contract's internal logic to potential adversaries. Malicious users can analyze the publicly visible code to identify vulnerabilities or weaknesses, which they can exploit for personal gain. This makes robust and secure development practices critical to anticipate potential attack vectors and mitigate risks.

Irreversibility The execution of a smart contract is irreversible, meaning that once initiated, its outcomes cannot be undone. Irreversibility ensures the reliability of smart

contracts as enforcement mechanisms, eliminating disputes over whether a contract's terms were executed correctly. At the same time, transactions resulting from flaws in the logic or from vulnerability exploitation cannot be rolled back. Developers must therefore ensure that contracts are carefully designed and thoroughly tested to avoid unexpected or undesirable consequences.

2.3.2 The Solidity Programming Language

Solidity is a high-level, Turing-complete programming language designed for writing smart contracts on the Ethereum blockchain. Because Solidity is Turing-complete, it allows for the implementation of any computable function, giving developers the flexibility to encode intricate business logic into their smart contracts. When developers write code in Solidity, it is compiled into *bytecode* that can be executed by the EVM.

The syntax of Solidity is influenced by JavaScript, however, it is statically and strongly typed. Solidity supports inheritance, polymorphism, and user-defined data types. Moreover, it introduces several novel constructs and keywords dedicated to smart contracts development. For example, Solidity includes keywords like `payable`, which designates functions that can receive Ether, the native cryptocurrency of Ethereum.

One of the peculiarities of Solidity is its gas model. Each operation in a Solidity contract consumes a certain amount of gas, which is a unit of computational effort. This gas mechanism is integral to Ethereum's design, as it prevents abuse of the network. Developers must be mindful of the gas costs associated with their code, as overly complex or inefficient code can lead to high execution costs.

2.3.3 The Smart Contract Storage Model

Every Ethereum smart contract features a permanent *storage* that retains data between transactions. This storage functions as an independent, read-write memory area holding the values of the contract's *state variables*, and each contract can exclusively access and modify its own storage. The storage acts like a *key-value database*, where each key corresponds to a storage *slot* number, and the *value* represents the content of that slot. The EVM rules governing how contract's state variables are laid out in long-term memory are referred to as **storage layout**. While the layout is complex, this Section only covers the key concepts, and more detailed information can be found in the official Solidity documentation[1].

A contract's storage is divided into 2^{256} slots of 32 bytes each, and space for state variables is pre-allocated at the time of contract construction: the variables are stored contiguously, starting from slot 0, and following the order in which they are declared. Ethereum optimizes storage efficiency, allowing variables to share slots when their data types and sizes permit. For example, consider the smart contract `C` shown in Figure 2.2. Each state variable is assigned an initial storage slot p sequentially (e.g.: variable `a` is at slot 0, `b` is at slot 1, and so on). Simple variables can generally be accessed from their initial storage slot p. However, for dynamic or complex data types (e.g., dynamic arrays, mappings, and structs) additional rules must be applied to determine where their data

[1] Layout of State Variables in Solidity: `https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html`

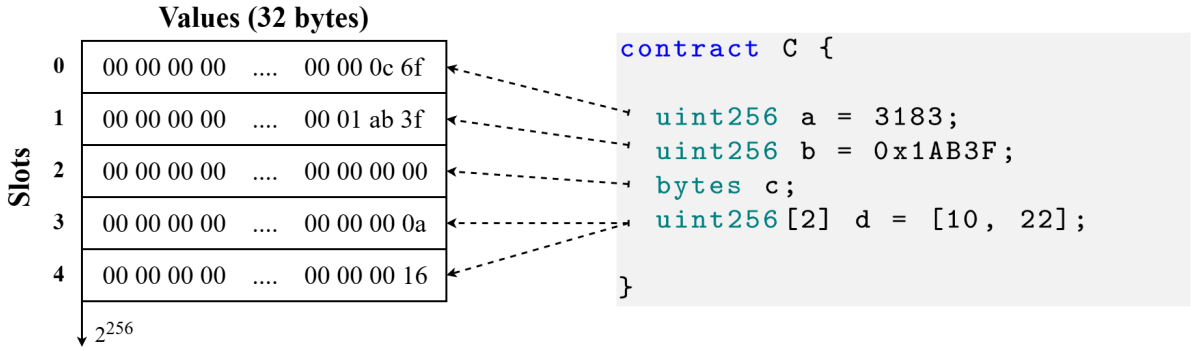


Figure 2.2: Example of Smart Contract Storage Layout

is stored. Dynamic arrays, for instance, have an initial slot that contains the number of elements, and the actual data slots are computed by hashing the slot number (i.e., using $keccak256(p)$). Mappings are more complex to extract, as they involve key-based slot computations to locate their elements.

Utilities like `hardhat-storage-layout`[2] support developers by displaying information about a contract's state variables and how they are laid out in storage. This can reduce the risk of human error when working with smart contracts and updating specific storage slots. However, these utilities only display the initial slot p where each state variable is located, offering a limited view of the storage data. For advanced techniques that rely on storage analysis (e.g., automated testing or state validation) more comprehensive methods are required to thoroughly explore the content of a contract's storage.

2.3.4 Upgradeable Smart Contracts

The main characteristic of a smart contract is its immutability, meaning its source code cannot be directly modified after deployment. In the case of a traditional (i.e., non-upgradeable) smart contract (Figure 2.3), the code can only be updated through a *contract migration* [99]. This is a process where a new contract (i.e., the new version) is deployed at a different address, starting with an empty storage. The state data from the original contract (i.e., the old version) is then migrated to the new one. Usually, an official announcement follows to make the community aware of the migration, allowing users, off-chain systems and other contracts to transition to the new address. This approach is not only inconvenient but also introduces complexities and costs due to the data migration process.

Upgrading a smart contract means switching the code executed at a given address while preserving its state, and without disrupting user interactions. An *Upgradeable Smart Contract (USC)* can be built using different patterns (e.g., the data separation pattern, the strategy pattern, and metamorphic contracts [99]), however the *proxy pattern* is the most widely used upgradeability mechanism on Ethereum. The idea behind the **proxy pattern** is to *decouple the storage* of a smart contract *from its logic*. This way, clients always interact with the same proxy contract, while the underlying logic can be replaced without losing any storage data.

[2]HardHat Storage Layout: `https://www.npmjs.com/package/hardhat-storage-layout`

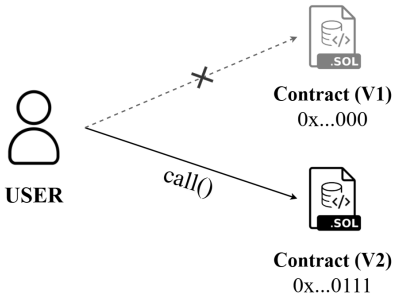


Figure 2.3: Standard Smart Contract

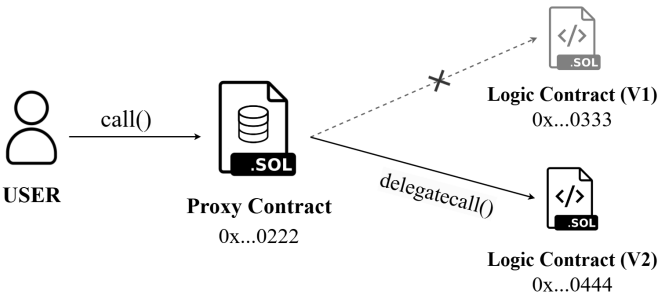


Figure 2.4: Upgradeable Smart Contract using the Proxy Pattern

As illustrated in Figure 2.4, a basic proxy architecture includes two components: a *proxy* contract and a *logic* contract (sometimes also referred to as *implementation* contract). All transactions issued by users target the proxy, which maintains the relevant storage data and balance of the dApp, as well as a mutable reference (i.e., an address) to a specific logic contract. When the proxy is invoked, it automatically reroutes the transaction to the referenced logic contract, which implements the DApp's business logic. This logic can therefore be replaced without re-deploying the proxy and incurring in costly state migrations. To achieve storage decoupling, the proxy pattern relies on two key mechanisms. Each transaction to the proxy is redirected to the logic contract using a `delegatecall`. This is a low-level Ethereum function that executes the logic of the called contract in the context of its caller. This means any operation performed by the logic contract will only affect the state of the proxy. Beyond the delegatecall, proxy contracts also require a function for controlling upgrades, allowing the address of the implementation used by the proxy to be changed with the address of a different logic contract. Generally, the upgrade function is built directly into the proxy with appropriate access control mechanisms to prevent unauthorized users from making changes.

CHAPTER 3

SOFTWARE QUALITY ASSURANCE

As software systems grow increasingly complex and critical in many domains, the importance of quality assurance cannot be overstated. Ensuring software quality is not solely a matter of executing tests but also requires systematic approaches to improve the effectiveness of the testing process itself and release more reliable code.

This chapter provides an overview of key quality assurance concepts in software development, with a particular focus on techniques relevant to smart contract testing, auditing and maintenance. Section 3.1 introduces fundamental software quality concepts and outlines the role of *Software Quality Assurance (SQA)*. In particular, Sections 3.2 and 3.3 introduce both *static* and *dynamic* SQA techniques, providing essential context for the thesis. Subsequent sections present those techniques that are most relevant to the specific Research Objectives of the thesis in more detail.

A core challenge in smart contract testing is *test adequacy assessment*, the focus of RO1. Sections 3.4 and 3.5 explore different approaches to evaluating the quality of test suites. Code coverage analysis, discussed in Section 3.4, is a widely used but often insufficient measure, particularly in the blockchain domain where it fails to capture deeper fault-detection capabilities. To address these limitations, Section 3.5 presents *mutation testing* as a more robust alternative. Additionally, Section 3.7 introduces *regression testing*, with a special focus on regression test selection, providing context on how this technique can help speed up mutation testing on evolving codebases.

Beyond assessing test adequacy, RO2 explores how mutation testing can support smart contract auditing processes. Given the high stakes associated with blockchain applications, developers frequently seek third-party audits before deployment. Section 3.2 introduces manual code review and static analysis, both of which are key activities performed by auditors. Section 3.6 introduces software auditing, with a focus on the smart contract domain. Understanding this process provides important context for discussing how dynamic techniques like mutation testing can complement static code inspection.

Finally, RO3 concerns *smart contract maintenance and upgrade practices*. In this context, effective regression testing techniques are essential to ensure backward compatibility

and avoid the introduction of new bugs. Section 3.8 introduces *Capture-Replay Testing*, an advanced technique that permits to validate software upgrades against real-world usage patterns.

3.1 Software Quality

In the world of software development, quality is a universal goal. Every stakeholder, be it developers or end-users, desires high-quality software. Software quality has been defined in various ways, but one widely accepted definition comes from the IEEE [154], which describes it as:

> **Software Quality:** the degree to which a system, component, or process:
>
> 1. *Conforms to specified requirements.*
> 2. *Meets customer needs or user expectations.*

Achieving high-quality software begins with well-defined requirements that state quality expectations. Without a clear understanding of what needs to be built, ensuring alignment with expectations is not possible. However, defining and measuring software quality is complex due to its multidimensional nature. Quality in software is not a single attribute, but a combination of *attributes* that together shape the degree to which it meets requirements and expectations. In practice, the priority of these attributes can vary significantly based on the software domain. A medical system might prioritize reliability over user-friendliness, while a consumer-facing mobile app might heavily optimize for usability. Because quality can mean different things to different audiences, organizations often adopt formal models or standards to structure their assessment. One widely recognized framework is the *ISO/IEC 25010*, which classifies quality attributes and allows teams to align on what high-quality entails for their product.

3.1.1 Software Quality Attributes

Software quality is evaluated using various attributes that reflect different aspects of the software's structure and behavior. The ISO/IEC 25010 [82] standard presents detailed quality models for computer systems and software products. The *product quality model*, shown in Figure 3.1 is composed of nine attributes (which are further subdivided into sub-attributes) that relate to quality properties of the products. The attributes and sub-attributes provide a reference model for the quality of the products to be specified, measured and evaluated. Quality attributes can be categorized according to different dimensions, but they are typically grouped into *static* and *dynamic*. These are summarized in Table 3.1.

Static Quality Attributes These pertain to properties that can be examined without executing the software, including the internal structure of the code, documentation clarity, and conformance to coding or architectural standards. Examples include attributes related to *maintainability*, such as *testability*, and *modularity*. These attributes are frequently assessed through *code reviews*, *static analysis tools*, and *conformance checks*

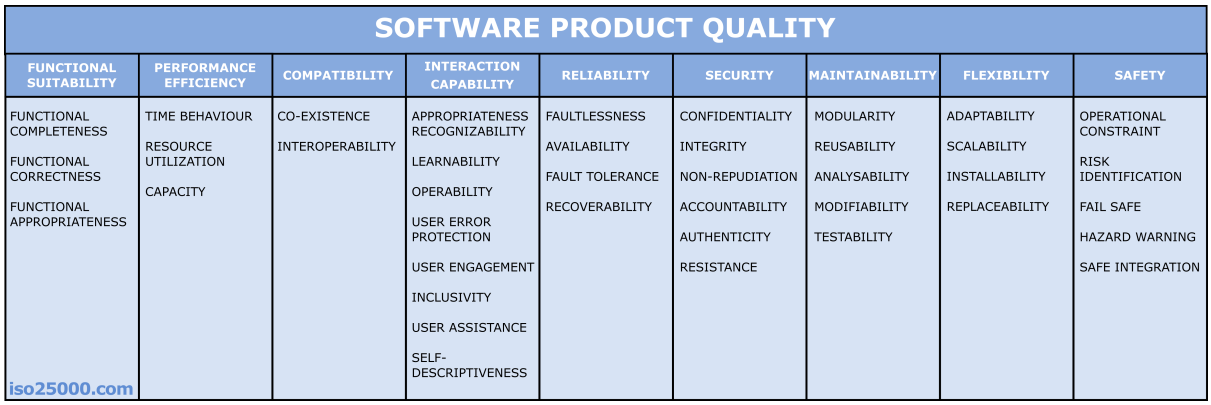


Figure 3.1: ISO/IEC 25010 Product Quality Model

Attribute	**Type**	**Description**
Functional Suitability	Static	The capability of a product to provide functions that meet needs of intended users when it is used under specified conditions
Compatibility	Static	The capability of a product to exchange information with other products, and/or to perform its required functions while sharing the same common environment and resources
Maintainability	Static	The capability of a product to be modified by the intended maintainers with effectiveness and efficiency
Flexibility	Static	The capability of a product to be adapted to changes in its requirements, contexts of use, or system environment
Performance efficiency	Dynamic	The capability of a product to perform its functions within specified time and throughput parameters and be efficient in the use of resources under specified conditions
Interaction Capability	Dynamic	The capability of a product to be interacted with by specified users to exchange information between a user and a system via the user interface to complete the intended task.
Reliability	Dynamic	The capability of a product to perform specified functions under specified conditions for a specified period of time without interruptions and failures
Security	Dynamic	The capability of a product to protect information and data, and to defend against attack patterns by malicious actors
Safety	Dynamic	The capability of a product under defined conditions to avoid a state in which human life, health, property, or the environment is endangered.

Table 3.1: ISO/IEC 25010 Static and Dynamic Software Quality Attributes

against known best practices. Investing in strong static quality helps reduce future maintenance costs and ensures that developers can quickly locate and fix defects.

Dynamic Quality Attributes These refer to the behavior of the software during execution. Dynamic attributes are the most visible to end-users and define their perception of the software's quality. Thus, dynamic quality attributes tend to receive significant attention in commercial and mission-critical software. These attributes are largely validated through *testing* (functional, performance, etc.) under representative or real-world conditions.

No single quality attribute determines the overall quality of software. Developers and tester must pay attention and evaluate both static and dynamic attributes to ensure that software not only meets functional requirements but also delivers a satisfying user experience. However, the *importance of each attribute depends on the context of the software's use*. For example, a business-critical system such as a smart contract managing financial transactions must prioritize reliability and functional suitability. The following Sections present processes and specific techniques that allow to maintain and improve the quality of software throughout its life-cycle. Ultimately, the choice of techniques and the extent to which they are applied depends on the context of the software, and the target quality attributes. However, combining static techniques with testing leads to a proactive Quality Assurance (QA) process that aligns with the multidimensional nature of software quality.

3.1.2 Software Quality Assurance

Software Quality Assurance encompasses a set of systematic activities designed to assess, monitor, and improve software development processes and work products, ensuring that they meet quality requirements and business objectives [154]. It includes the processes, practices, and techniques aimed at preventing defects rather than just detecting them, ensuring that software functions as intended and adheres to industry standards.

While the previous sections introduced the concept of software quality and its key attributes, the rest of this chapter shifts focus to *how* these attributes can be practically evaluated and maintained throughout the software life cycle. Specifically, it explores two fundamental quality assurance methodologies: static analysis and dynamic analysis. These two approaches can be broadly distinguished as follows:

> Static Analysis: "Examine the source code and possibly complain."
> Dynamic Analysis: "Execute the program and possibly complain."

Static analysis evaluates software without execution, identifying potential defects such as syntax errors, security vulnerabilities, and code maintainability issues at an early stage. Dynamic analysis, on the other hand, tests software during execution, validating its functionality, performance, and security in real-world scenarios. It includes techniques like testing and runtime verification, which help uncover issues that static methods may overlook. These two techniques are inherently complementary. While static analysis provides early-stage defect detection, reducing the cost of fixing issues later, dynamic

analysis ensures that the software behaves correctly in real-world conditions. Relying on only one approach can leave gaps in the quality assurance process, making a combination of both essential for robust software development. The following sections will explore static and dynamic analysis techniques in more detail.

3.2 Static Quality Assurance Techniques

Static QA techniques (sometimes referred to as static testing) focus on examining and verifying the software's structure *without* executing the code. They aim to catch issues early in the development process, reducing the risk of defects propagating into later, more expensive stages. Beyond the code itself, QA teams may review architecture documents and design specifications to ensure they conform to organizational standards or formal models. However, this section only focuses on the analysis of the code itself. In this sense, there are two main techniques that can be applied: manual code reviews, and static analysis. Both of these are key activities in smart contract auditing [2], which is discussed in more detail in Section 3.6.

3.2.1 Manual Code Review

A manual code review is a process where one or more designated reviewers examine the source code to identify issues related to correctness, maintainability, security, and adherence to coding standards. Reviews can range from informal examinations to more structured code inspections, where checklists and predefined roles guide the process. During a formal inspection, developers, testers, or external auditors systematically analyze the source code to detect bugs, inefficiencies, security vulnerabilities, and deviations from best practices or design specifications. Code that undergoes rigorous review is significantly less likely to contain vulnerabilities and is often more readable and maintainable compared to unreviewed code [25]. However, manual reviews have limitations, as they can be time-consuming and expensive. In the context of smart contract auditing, manual reviews play a critical role in uncovering bugs and vulnerabilities that automated tools may overlook. This process is described in more detail in Section 3.6.

3.2.2 Static Code Analysis

Automated static code analysis is a technique used to analyze source code without executing it. This approach is widely employed to identify various types of issues, including syntax errors, inefficient code, security vulnerabilities, code smells, and violations of predefined coding rules. Unlike manual code reviews, which rely on human expertise, static analysis tools apply predefined rules and patterns to the code, offering a faster and more scalable way to identify issues across large codebases. The static analysis process involves parsing the source code, applying rule-based checks, and performing more advanced techniques such as data flow and control flow analysis to detect deeper issues. An appropriate configuration of static analysis tool helps developers focus on relevant warnings and address underlying issues before the software undergoes further testing [132].

3.3 Dynamic Quality Assurance Techniques

Dynamic QA techniques evaluate software *during execution* to measure its real-world behavior. They aim to validate that the software meets functional requirements and user expectations under a variety of conditions. Dynamic QA involves different shades of **software testing**, a systematic process that evaluates the behavior of a software product to acquire confidence in its quality. Section 3.3.1 introduces the fundamentals of software testing, whereas Section 3.3.2 provides an overview of existing test types.

3.3.1 Software Testing

Errors are an inherent aspect of human activity, manifesting in our thoughts, actions, and the various products that emerge from these processes. In the realm of software development, errors can have significant implications. These cause the software to behave unexpectedly or incorrectly, potentially compromising its functionality and reliability.

Objectives of Testing

Software testing is the process of *evaluating the behavior of a software product* under controlled conditions to *acquire confidence in the correctness of its implementation*. Although testing can never guarantee the absence of faults, it can increase our confidence in their absence. More specifically, software testing has two objectives:

(1) *Finding faults*: Testing aims to identify faults in the software. These manifest as deviations from expected behavior or functionality.

(2) *Ensuring compliance with requirements:* Testing aims to ensure that all features work as expected and performance metrics are achieved.

Clearly, mission and business critical software demands a higher degree of confidence in the **correctness** and **reliability** of their implementation. As said in Section 3.1, both are key attributes of software quality, but there is a key distinction between the two.

Correctness aims to establish that a program, under all conditions and for all possible inputs, produces the intended outcomes. Correctness is a binary attribute: a program is either correct or incorrect. While correctness is an ideal goal, achieving it through testing is almost always impossible due to the complexity of most real-world software systems. That's why, in practice, testing is rarely aimed at demonstrating correctness, but at *detecting faults* in the program. While a program free of known faults is not guaranteed to be correct, its *quality improves* with thorough testing and defect resolution.

Reliability, on the other hand, is a statistical attribute that quantifies the likelihood of a program performing its intended function without *failure* over a specific period and under defined conditions. Reliability is inherently probabilistic because it acknowledges the practical limitations of exhaustive testing. Instead of attempting to prove that a program is error-free, reliability focuses on how likely it is that the program will function correctly in real-world scenarios.

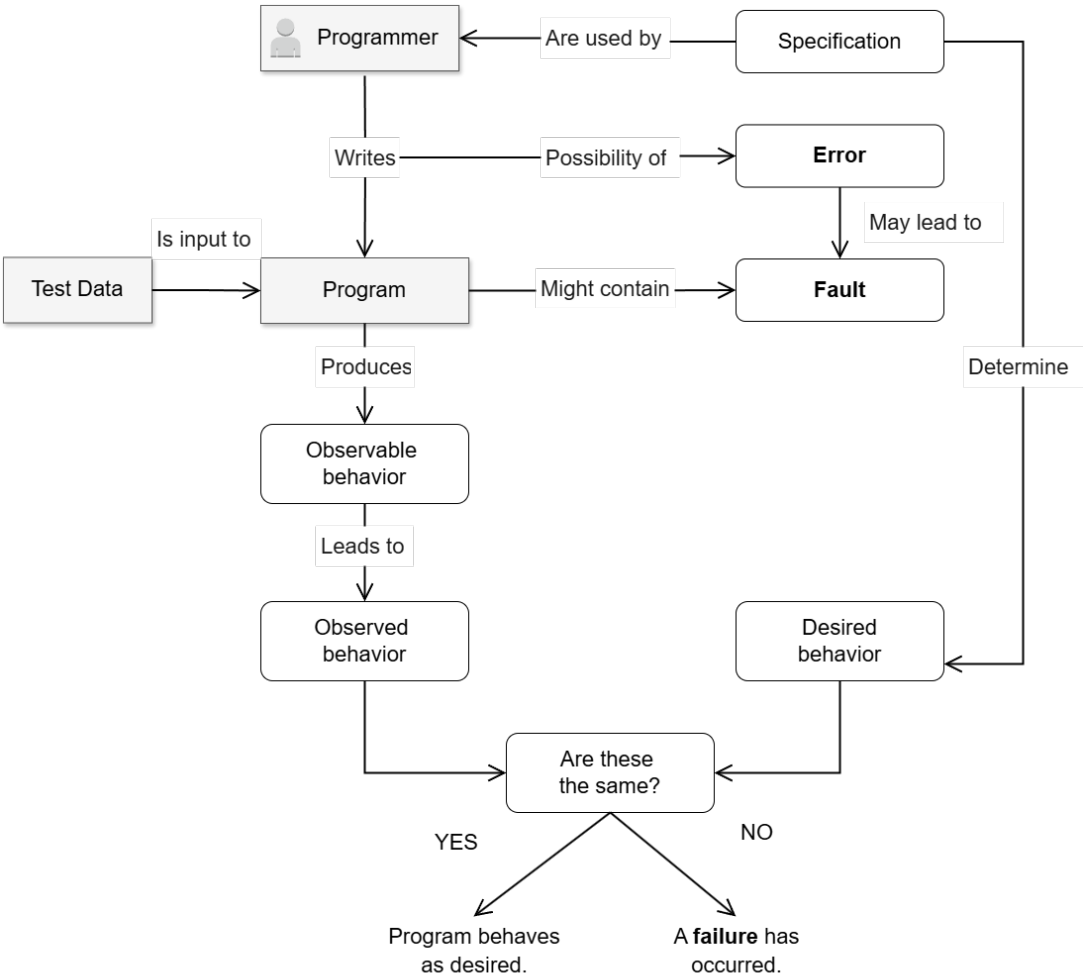


Figure 3.2: Errors, Faults, and Failures in the Process of Programming and Testing

Genesis of Software Failures

To clarify the role of testing, Figure 3.2 illustrates the genesis of software failures.

An **error** is a human mistake that may occur in any phase of software development, leading to a mismatch between the software and its specification. While there is no universally accepted and definition of the term *error*, Mathur [111] refers to it as to a mistake made during the process of writing or designing a program.

When an error is encoded into the software, it becomes as a **fault** (commonly referred to as a defect or bug). A fault represents one or more underlying errors that have been introduced into the code.

Upon execution of a faulty piece of code, a **failure** occurs. A failure is the observable manifestation of incorrect behavior, where the software's output deviates from the expected behavior as per its specification. For instance, a programmer might misinterpret the requirements and consequently write incorrect code. Although the fault exists in the code, it might remain dormant until a specific execution path triggers it.

As shown in Figure 3.2, there is a separation between the software's *observable behavior* (what the software is designed to do) and its *observed behavior* (what the software actually does during execution). Failures are often detected by analyzing the observed behavior and identifying discrepancies from the expected outcomes.

3.3.2 Types of Testing

A **test type** is a group of test activities based on specific test objectives aimed at specific characteristics of a component or system. One of the most widely recognized classifications of test types [83] is provided by the International Software Testing Qualifications Board (ISTQB). As shown in Figure 3.3, the ISTQB classification defines the following types of testing: *functional*, *non-functional*, *structural* and *change-based*.

Functional Testing

Functional testing verifies that the software behaves according to specified requirements by evaluating its features and expected outputs. Functional testing is *specification-based* (also known as *black-box* testing), as it assesses the behavior of the software without considering its internal structure. This ensures that the software meets expectations by focusing on inputs and expected outputs rather than the underlying code. Thorough functional testing improves the likelihood of releasing correct and reliable software. Functional testing is commonly classified into **test levels**, each targeting a different scope within the software development life-cycle. These levels ensure that defects are identified at the most appropriate stage, reducing the cost and complexity of fixing them later.

1) *Unit Testing* focuses on individual components or functions to verify that they work correctly in isolation;

2) *Integration Testing* validates multiple units or components together to ensure that they work as a group and that interactions between them are functioning as expected;

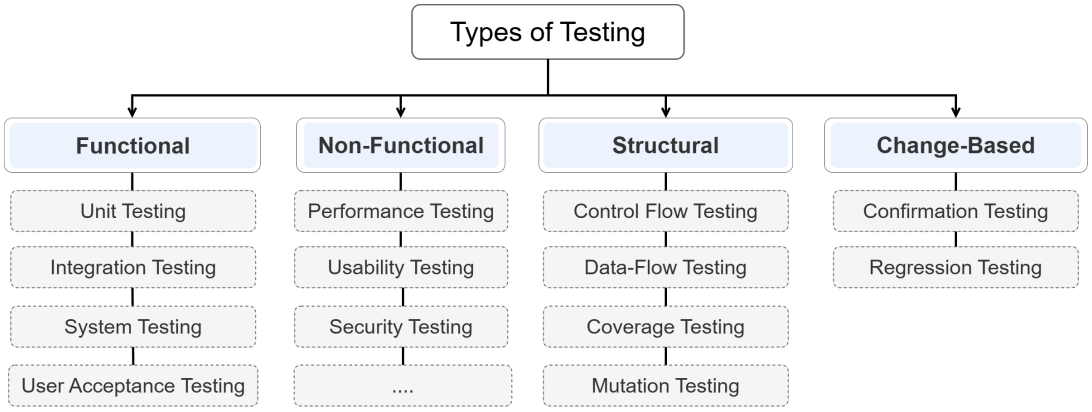


Figure 3.3: ISTQB Testing Types Classification

3) *System Testing* evaluates the complete application as a whole, verifying that it meets functional requirements and behaves as expected;

4) *User Acceptance Testing* is conducted from the end-user's perspective to validate that the software aligns with business goals and user needs before deployment.

Non-Functional Testing

Like functional testing, non-functional testing is specification-based and is performed at all test levels. However, rather than validating *what* the system does, non-functional testing evaluates *how well* it does it. This testing type assesses software behavior against predefined criteria without examining internal code structure, but it focuses on non-functional quality attributes. It assesses aspects such as *performance*, *security*, *usability*, and *scalability*, which are critical for real-world deployment. For example, performance and load testing measure response times and system scalability, and are particularly critical for applications with large user bases or stringent real-time requirements. Security testing uncovers vulnerabilities that attackers might exploit, and usability testing identifies user-experience issues that static reviews cannot catch, improving accessibility and ease of use. This thesis mostly focuses on functional, structural and change-based testing techniques, therefore non-functional testing will not be further discussed.

Structural (or White-Box) Testing

The goal for structural testing, also known as *white-box testing*, is to ensure that the internal components of a program are working properly. Unlike specification-based testing, which validates externally observable behaviors, structural testing focuses on the internal implementation of the software. Since it requires knowledge of the program's internal workings, it is often carried out by the developers themselves. In particular, the developer creates test cases that exercise these structural elements, such as loops and branches, to determine if defects exist in the program. The term exercise is used in this context to indicate that the target structural elements are executed when the test cases are run. By exercising all of the selected structural elements the tester hopes to improve

the chances for detecting defects. Structural testing is commonly applied at the *unit* and *integration* levels, where developers validate individual components and their interactions. A variety of techniques exist to target different structural attributes and to increase confidence in the software's reliability. The most widely used methods are outlined below.

Control Flow Testing systematically examines the paths a program can take during execution. Typically, a control flow graph is constructed to represent the code's loops, branches, and decision points. Test cases are then created to traverse each significant path in the graph. Control flow testing helps to uncover faults that may not surface if only a subset of paths were tested.

Data-Flow Testing focuses on how data is defined, used, and propagated throughout the code. It tracks variable definitions and their subsequent uses to uncover issues such as uninitialized variables, redundant definitions, or incorrect assignments. Data flow testing is particularly effective at exposing defects related to the handling of data values, such as premature overwrites.

Coverage Testing systematically defines tests that exercise specific structural elements within the code (e.g., statements, branches). Ensuring that such elements are executed at least once increases confidence that major structural components of the software have been validated. At the same time, coverage analysis reveals which areas of the code remain untested or under-tested, helping to derive more effective test cases. Coverage analysis is widely used in practice, thus Section 3.4 further discusses its different metrics and limitations.

Mutation Testing focuses on defining tests that have good fault-detection capabilities. It injects minor faulty modifications, known as *mutants*, into the source code (for example, by changing arithmetic operators). The test suite should fail on these altered versions to indicate that it can catch the introduced bugs. Test cases can be therefore refined based on undetected faults to increase their effectiveness. Mutation testing is a central topic of this thesis, therefore it is comprehensively discussed in Section 3.5.

Change-Based Testing

Change-Based testing is a type of testing initiated by modification to a component or system, which could stem from defect fixes, environmental changes, functional modifications, or the introduction of new features. Its objective is ensuring that changes have been successfully implemented and that their inclusion has not resulted in unexpected regressions. Two main testing techniques are employed to address these concerns.

Regression Testing verifies that recent changes do not negatively impact the system. For example, fixing a miscalculation bug in the weekend rate calculation of a timesheet system might inadvertently introduce faults in standard day rates. Thus, it is important to systematically re-run previously executed tests to detect unintended consequences. Section 3.7 will discuss regression testing in more detail.

Metric	What it Measures	What it Provides
Statement Coverage	The percentage of executable statements that are exercised by the test suite. Each statement must be executed at least once.	Basic insight into which parts of the code are tested but does not account for different control flows.
Branch Coverage	The percentage of branches (decision points in the code) that are exercised. A branch occurs at conditional statements (such as `if` statements).	Ensures that both true and false outcomes are tested.
Decision Coverage	Whether all possible outcomes of conditional expressions are covered by tests.	Similar to branch coverage but focuses on covering all logical decisions within a condition.
Path Coverage	The percentage of unique execution paths through the program that are exercised by tests.	Ensures that each possible route the program can take is tested at least once.

Table 3.2: Code Coverage Metrics

Confirmation Testing (or Re-Testing) ensures that an introduced change meets its intended specifications. In the same timesheet example, confirmation testing would validate that the weekend rate calculation bug has been successfully corrected. Re-testing typically involves re-executing test cases that previously failed, focusing solely on validating resolved issues rather than detecting new regressions.

3.4 Code Coverage Analysis

Testing activities are essential for gaining confidence in the correctness of a software product. However, the level of confidence derived from testing depends on the quality, or *adequacy*, of the test suite being used. Adequacy is a quality metric that must be evaluated according to predefined criteria. Adequacy criteria serve as a framework to determine which internal elements of a program should be tested and guide the selection of test data. Formally, a test adequacy criterion is a *stopping rule* [190], a guideline that helps testers decide when testing has been conducted to a sufficient degree, and whether they can reasonably move forward. Adequacy criteria are frequently employed in structural testing, which was introduced in Section 3.3.2. They use logic and control structures (in code coverage), data paths (in data flow testing), or intentional faults (in mutation testing) as the focal points of the adequacy evaluation [206].

Among these, *code coverage* is one of the most widely used adequacy criteria in both industry and academia. Since it indicates the percentage of the codebase exercised by the test suite, coverage reveals which parts of the code remain untested and therefore might hide defects. There exist a variety of coverage metrics, each of which captures different structural elements of the code. Some focus on whether every statement is executed at least once, others on whether all branches in conditional statements are tested, and still others on whether specific paths or data dependencies within the code are covered. These dimensions of coverage, illustrated in Table 3.2, reflect differing levels of rigor, from basic

forms such as statement coverage to more advanced approaches like path coverage. One reason code coverage is widely used in practice is that it is intuitive, widely supported by testing frameworks across most languages, and easy to collect.

Despite its practical usefulness, code coverage suffers from important limitations that reduce its ability to reflect true test effectiveness. High coverage levels do not automatically translate into high defect detection rates, a conclusion supported by many empirical studies [81, 182]. Code coverage is highly helpful for identifying untested portions of a system, but it is not always a reliable indicator of whether the tests thoroughly check the program's logic. While low coverage means a lot, achieving high coverage does not actually tell us much about the quality of a test suite. Furthermore, more complex forms of coverage, such as path coverage, do not necessarily guarantee meaningfully better results in terms of finding defects, as these measures do not inherently capture semantic correctness or address how the software's functional requirements are exercised.

```
1  // A simple function that returns a greeting for a given name
2   function greet(name) {
3     return "Hello, " + name + "!";
4   }
5
6  // Test suite for the greet function
7   describe("greet function", () => {
8   it("should greet a person by name", () => {
9     // All statements of the greet function are executed
10    greet("Alice");
11  }
12 });
```

Listing 3.1: JavaScript Code Achieving 100% Statement Coverage with No Assertions

Listing 3.1 is a simplified example in JavaScript that illustrates how one can achieve 100% code coverage without performing any actual checks. Despite the fact that this test calls the function and executes all lines of code, it does not actually verify whether the function's output matches the expected result. Coverage tools will mark all statements as covered, even though the effectiveness of the test in terms of defect detection is null. To mitigate these shortcomings, advanced approaches such as mutation testing can be employed. Mutation testing, which is better discussed in the next Section, complements coverage by injecting artificial defects into the code and checking whether the tests detect them. This type of assessment offers a more indicative estimation of how well the test suite can catch real-world defects, providing a deeper insight into the test suite's quality.

3.5 Mutation Testing

Mutation testing is a white-box, fault-based technique for evaluating and enhancing the quality of a test suite. It works by introducing small changes, or *"mutations"*, to the code and then checking whether the test suite can detect the introduced faults. This helps evaluate the effectiveness of the test suite, ensuring that it can detect subtle errors,

rather than simply achieving high coverage. Mutation testing complements traditional code coverage metrics by assessing whether the tests are not only extensive but also capable of uncovering real defects. Mutation testing is based on two *key hypotheses*:

(1) *Competent Programmer Hypothesis*: This hypothesis states that developers tend to write code that is "close to correct," meaning that most errors they introduce are small syntactic mistakes.

(2) *Coupling Effect Hypothesis*: This hypothesis states that simple faults are strongly coupled to complex faults.

According to these hypotheses, small syntactic changes (mutations) approximate the subtle mistakes developers are likely to introduce. If tests can detect these small faults, they are also likely to detect more complex bugs. Thus, mutation testing deliberately introduces minor alterations to the code and checks whether the test suite identifies them.

3.5.1 Mutation Testing Process

Figure 3.4 illustrates the mutation testing process. Mutation testing begins with a program P and a corresponding test suite T to be evaluated. The process involves generating **mutants**, which are slightly modified versions of the original program P. These mutants are produced by applying a set of predefined **mutation operators**, each representing a common programming error. The test suite T is then run against each mutant to evaluate whether the introduced fault is detected by the tests.

Once mutants are created, the test suite is executed against each one to determine if the tests can detect the injected fault. If at least one test case in T detects the fault, the mutant is considered *killed*. If no test case identifies the fault, the mutant is considered *live*, indicating a potential weakness in the test suite. The percentage of (valid) mutants killed by T, called **Mutation Score (MS)**, represents the adequacy value of the test suite. A higher MS indicates that the test suite is more effective at detecting faults. Developers can iteratively improve the test suite by analyzing the live mutants and writing new tests to cover previously untested scenarios. This process generally goes on until the adequacy threshold set by the tester is satisfied by the MS.

3.5.2 Mutation Operators

A key element in every mutation testing approach is a set of fault-injection rules called *mutation operators*. These operators define how the source code is altered to create *mutants*, and are designed to simulate various types of fault stemming from human error. In general, programming errors fall into three main categories: *logical*, *semantic*, and *syntactic*, but only the former two are of interest in mutation testing.

A *logical error* is caused by a flaw in the programmer's logic during the elaboration of a solution. A program that contains a logical fault is still syntactically correct, but its behavior deviates from the intended logic, potentially causing incorrect outputs. An example of logical error is encoding the wrong logic in a conditional statement.

Instead, a *semantic error* originates from a misunderstanding or misuse of language-specific features. Even if a solution is logically sound, developers may make mistakes due

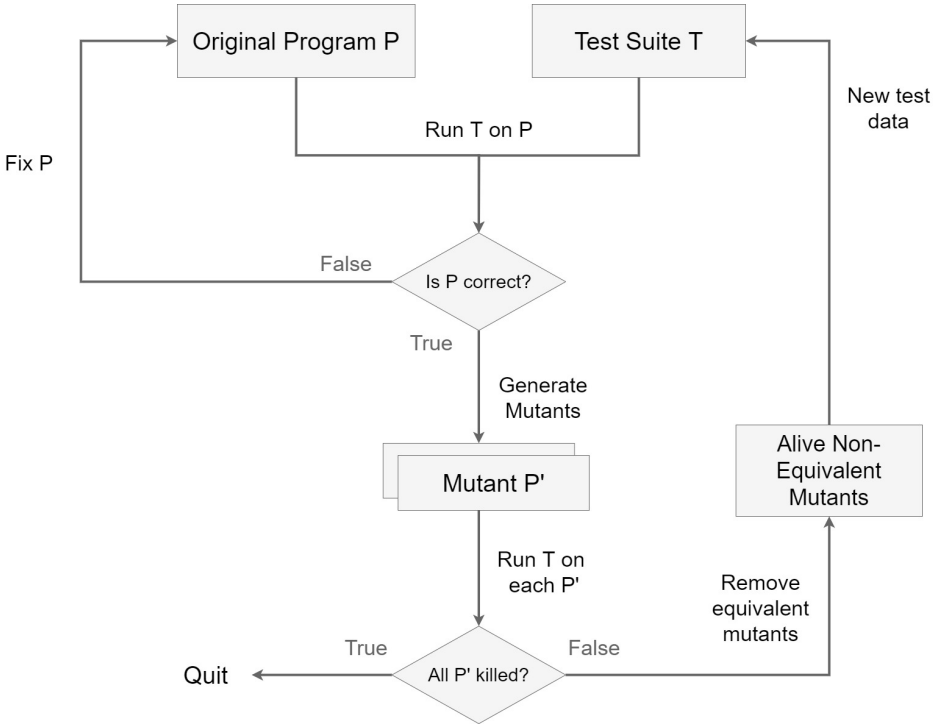


Figure 3.4: Mutation Testing Process

to unfamiliar syntax, insufficient documentation, or frequent language updates. Such errors can remain undetected by the compiler and manifest only at run-time or during manual inspection. For example, consider the `transferOwner` Solidity function in Listing 3.2. The owner of the contract can invoke this method to transfer ownership to another address (i.e., `newOwner`). The `onlyOwner` modifier attached to the function prevents unauthorized users from performing this operation. Here, a mutation operator removes the modifier from the function (The original statement is marked with ⊙, whereas the mutated statement is marked with ⋆). This change simulates the developer accidentally omitting the security check, allowing unauthorized users to perform actions they shouldn't be able to.

```
1 ⊙ function transferOwner(address owner) public onlyOwner {
2 ⋆ function transferOwner(address owner) public ~~onlyOwner~~ {
3     _transferOwnership(owner);
4 }
```

Listing 3.2: Example of Mutant for the Solidity language

Lastly, a *syntactic error* is caused by code that does not respect the syntax of the language. Since Solidity is a compiled language, syntactic errors are always detected at compile time. Injecting syntactic faults as mutants provides no benefit because they cannot be executed. instead, they become *uncompilable mutants* that must be discarded, causing unnecessary overhead.

Dimension	Definition and Classification
i) Validity	
Valid	A mutant that can be associated with a *pass* or *fail* test outcome.
Killed	A mutant for which at least one test gives a *fail* outcome.
Live	A mutant for which all tests give a *pass* outcome.
Invalid	A mutant that cannot be associated with a *pass* or *fail* test outcome.
Uncompilable or Stillborn	A mutant that does not compile.
Timed-out	A mutant that exceeds the maximum test run-time.
ii) Equivalence	
Equivalent	A mutant that is behaviorally identical to the original program.
Non-Equivalent or Killable	A mutant that is behaviorally different from the original program.
Redundant	A mutant that is behaviorally identical to another mutant.
Non-Redundant	A mutant that is behaviorally different from all other mutants.
iii) Subsumption	(as defined by Kurtz et al., [94])
Subsuming	A mutant m1 subsumes another mutant m2 if, whenever m1 is detected by a test, m2 is also detected.
Subsumed	A mutant m2 is subsumed by another mutant m1 if, whenever m1 is detected by a test, m2 is also detected.
iv) Productivity	(as defined by Petrovic et al., [139])
Productive	A mutant that elicits an effective test or whose analysis advances knowledge and code quality.
Unproductive	A mutant that represents a futile test goal and whose analysis does not advance knowledge and code quality.

Table 3.3: Classification of Mutants in Mutation Testing Based on Various Dimensions

3.5.3 Classification of Mutants

Mutation testing operates by introducing small mutations into a software artifact. However, not all mutants are created equal, and a careful classification helps guide more efficient testing strategies. This section outlines the fundamental categories of mutants along four (potentially orthogonal) dimensions: *validity*, *functional equivalence*, *subsumption* and *productivity*. Table 3.3 summarizes these categories along with a brief description.

1) Validity

First, mutants can be classified based on their validity. A **valid** mutant is a syntactically correct variation of the original program that can be *compiled* and *executed*. Thus, it can be associated with a valid test outcome (i.e., *pass* or *fail*) and is suitable for evaluating the adequacy of the test suite. Instead, an **invalid** mutant is unusable in further analyses. Invalid mutants represent an overhead for the mutation testing process, as they waste computational resources without providing useful insights about the quality of the tests. Based on this dimension, we can define:

Uncompilable or Stillborn Mutant: A mutant that does not compile.

Timed-Out Mutant: A mutant that takes too long to execute and is terminated by a pre-set time limit.

Listing 3.3 shows an example of uncompilable mutant for the Solidity language. Here, a mutation operator replaces the `public` visibility keyword of the `deposit` function with `private`. In the general case, this mutation is perfectly valid. However, since the `deposit` function is declared as `payable`, it must be able to receive ether and thus it must have either `public` or `external` visibility.

```
1 ⊙ function deposit(uint256 amount) payable public {
2 ⋆ function deposit(uint256 amount) payable private {
3     // deposit logic  ...
4   }
```

Listing 3.3: Example of Uncompilable Mutant

Valid mutants that successfully compile and executed, are therefore classified based on the final test outcome.

Killed Mutant: A mutant that is detected by at least one of the existing tests (i.e., the tests fail when run on the mutant).

Live Mutant: A mutant that survives all tests, meaning that the test suite does not fail even though the code is mutated. This often indicates a potential testing gap.

2) Functional Equivalence

Valid mutants can be categorized based on their functional behavior with respect to the original program or other mutants:

Non-Equivalent or Killable Mutant: A mutant that behaves differently from the original program for at least one input. Non-equivalent mutants are also known as killable, as there exists at least one test case can be derived to detect the behavioral change in the mutant.

Equivalent Mutant: An equivalent mutant is a mutant that behaves identically to the original program for all possible inputs. This means it cannot be distinguished from the original program because the change does not affect the program's output or behavior.

```
1 contract CrowdfundingCampaign {
2 ⊙ uint public constant MIN_DURATION  = 60 * 60 * 1 ;
3 * uint public constant MIN_DURATION  = 60 * 60 / 1;
4     ...
5 }
```

Listing 3.4: Example of Equivalent Mutant

Equivalent mutants are always *live* upon testing and they decrease the reliability of the Mutation Score because they are mislabeled as faulty programs. Listing 3.4 shows an equivalent mutant generated from a real-world Smart Contract. Here, a mutation operator replaced the mathematical operator * with / . However, since the right-hand operand is 1, the value of the `MIN_DURATION` constant is not affected.

Considering its relationship with other mutants, a mutant can also be classified as:

Non-Redundant Mutant: A mutant that is distinct from other mutants in its behavior or the type of fault it introduces. Killing this mutant requires a unique test case or provides unique insight into the quality of the test suite.

Redundant Mutant: A mutant that is functionally equivalent to another mutant, meaning it does not provide any additional value for assessing the test suite's ability to detect faults. If a test case can kill one mutant, it can also kill the redundant mutant without requiring any new or additional test cases.

For example, let's consider the two mutants applied to the return statement of `isPositive` in listing 3.5. Although they apply different mutations, the same test case for `num` = 0 will kill both of them. A redundant mutant does not contribute new information because it doesn't introduce new failure conditions for the test suite to detect.

```
1 contract NumberChecker {
2   function isPositive(int256 num) public returns (bool) {
3     ⊙ return num > 0;
4     * return num >= 0;
5     * return num > -1;
6   }
7 }
```

Listing 3.5: Example of Redundant Mutants

3) Subsumption

Subsumption is used to describe the relationship between (valid) mutants based on the test cases required to detect them. As defined by Kurtz et al., [94], a subsumption relation between a mutant *m1* and mutant *m2* exists when, if m1 is detected by a test, m2 is also detected. Thus, we can define:

Subsuming Mutant: A mutant whose detection (killing) implies the detection of one or more other mutants. In other words, if a test case can kill the subsuming mutant, it will also necessarily kill the subsumed mutants.

Subsumed Mutant: A mutant that is dominated by a subsuming mutant. If a test case kills the subsuming mutant, it will automatically kill the subsumed mutant as well.

Generally, the subsuming mutants are more influential in evaluating the test suite. If a test case kills the subsuming mutant, it will automatically kill the subsumed mutant as well. However, it's possible that a specific test case kills a subsumed mutant but does not kill the subsuming mutant. This implies that subsuming mutants are generally harder (or at least not easier) to kill. As such, they are more influential in evaluating the strength of a test suite, as their detection implies the detection of one or more other mutants. On the other hand, subsumed mutants are considered less informative in this context. Once a test suite is able to kill a subsuming mutant, the corresponding subsumed mutants add no additional insight into the effectiveness of the test suite. Therefore, subsumed mutants can increase computational overhead without contributing new information. As a result, subsumption analysis is often employed to reduce the number of mutants and focus on those that provide the most meaningful evaluation of test adequacy.

4) Productivity

Traditionally, mutants are classified as either killable (can be detected by the tests) or equivalent (cannot be distinguished from the original program because they have no observable effect). However, Petrovic et al. [139] observed that these classifications are not applicable in real-world settings. The authors introduce the concept of productive mutants, which refines the traditional classification by considering the practical value of a mutant's impact.

Productive Mutant: A mutant that 1) is killable and elicits an effective test, or 2) is equivalent but its analysis advances knowledge and code quality.

The distinction between productive and unproductive mutants is inherently qualitative, depending on the developer's judgment. For killable mutants, productivity is often clearer. A mutant is productive if it encourages the creation of a test that captures a previously untested behavior. However, for equivalent mutants, the distinction can be less obvious and context-dependent. For instance, a productive mutant might reveal error masking or redundancy in the code. It might also highlight ambiguities or faults, prompting the developer to refactor or simplify the code.

3.5.4 Limitations of Mutation Testing and Cost Reduction

Mutation Testing is a powerful technique for assessing the adequacy of a test suite. However, it suffers from several limitations that hinder its practical usability. One significant challenge is its *computational cost*, as the mutation analysis process is highly resource-intensive. Generating a large set of mutants and executing them against the test suite

is time-consuming and computationally expensive. To mitigate this issue, researchers have explored two primary approaches: reducing the *number of generated mutants* or *optimizing the cost of mutant execution.* However, these techniques must be applied carefully, as they can impact the effectiveness of the final test suite. Another major issue of mutation testing is the *manual effort* required from users, particularly in interpreting live mutants to derive new or improved test cases and discarding equivalent mutants. Automated detection techniques help, but equivalent mutant detection is fundamentally *undecidable problem*, thus their identification often requires additional human effort for filtering. Over the years, several techniques have been proposed to minimize both the generation of equivalent mutants and the manual effort required for their identification.

Reducing Mutants Techniques that reduce the number of mutants to be generated, tested and inspected help to reduce the computation cost of the evaluation and the manual effort required for inspection. Jia and Harman [86] identified four main techniques to reduce the number of mutants. *Mutant Sampling* randomly selects a subset of random mutants from the full set, assuming that its mutation score approximates that of the entire set. Instead of selecting mutants randomly, *Mutant Clustering* [77] algorithms group mutants based on their ability to be killed by similar test cases. A representative subset is then chosen from each cluster. *Selective Mutation* limits the set of mutation operators to those empirically found to be the most effective [201]. Instead, *Higher Order Mutation Testing* [85] combines multiple mutations to create higher-order mutants, reducing the number of total mutants. This however, can introduce masking effects when both mutations that would survive and mutations that would be killed are combined in the same mutant [91].

Reducing Execution Costs Techniques that optimize mutant execution time can reduce the time costs for the assessment. *Weak mutation testing* requires a test to reach a mutated statement and cause a state change, whereas *strong mutation testing* further requires the error to propagate to the program output. Thus, weak mutation testing is generally more computationally efficient [91]. Parallel processing also played a significant role in advancing mutation analysis. The concept of locally parallelizing mutation analysis was recognized since the early 90s [125]. More recently, the approach was thoroughly evaluated [110], extended across a variety of mutation testing tools, languages, and domains ([98, 37]), and brought to the cloud ([114]).

Detecting Equivalent Mutants Several strategies were proposed to reduce equivalent mutants, some of which overlap with mutant reduction techniques. *Selective Mutation* can help by choosing operators that empirically show low equivalent mutant generation rates [201]. Combining multiple mutations reduces the likelihood of generating equivalent mutants with *Higher Order Mutation Testing* also reduces the probability of generating equivalent mutants. The *HDom (50%)* technique, which selects a mix of first-order and higher-order mutants, was shownn to significantly reduces equivalent mutants [91]. However, due to its simplicity and ease of integration, the most widely used technique for detecting equivalencies is the *Trivial Compiler Equivalence (TCE).* Introduced by Papadakis et al. [133], this technique leverages compiler technology to compare mutant

binaries with the original program. Identical binaries indicate equivalent mutants, allowing up to 30% of equivalent mutants to be automatically discarded.

3.6 Software Audit

Software audits are commonly performed in high-stakes industries, such as finance, healthcare, and blockchain, where software failures can lead to significant financial or security repercussions. The IEEE defines a software audit [78] as:

> **Audit:** *An independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements, or other criteria.*

An audit is not merely a code review, but a structured, multi-layered process that combines both manual and automated analysis techniques to ensure that the software meets its intended specifications and security requirements. Auditing has become an integral part of the smart contract development life-cycle due to the high stakes involved and the adversarial nature of blockchain environments [130, 39]. Pre-deployment audits are an essential step in ensuring that smart contracts are resilient against attacks and unintended behaviors. According to a study [180] by Chainalysis, 2022 saw the highest amount of value stolen from smart contracts, a phenomenon that was mitigated with the increasing reliance of development teams on audit companies.

Private audits are conducted in a confidential setting by security firms like *Quantstamp*, *Hacken* and *Certik*, while public audits involve multiple independent security researchers competing to identify vulnerabilities, often through platforms like Code4rena [43]. Smart contract audits are both *time-sensitive* and *resource-intensive*, often requiring a dedicated team of professionals over the course of weeks for a standard protocol. The cost of an audit varies widely depending on the size and complexity of the codebase, with prices ranging from $5,000 for small-scale contracts to over $60,000 for large and intricate protocols [181]. More complex projects may also necessitate extended review periods, particularly when dealing with intricate financial mechanisms, or complex cross-chain interactions.

The Auditing Process

Smart contract audits typically follow a structured process that involves both automated and manual analysis. As an example, Figure 3.5 shows the auditing process at *Quantstamp* [144], a leader blockchain security company. Once a protocol engages the audit firm, the process begins with an initial assessment, where auditors review the contract's specifications, intended functionality, and system architecture. This step provides auditors with a high-level understanding of the smart contract's purpose and intended behavior. Each auditors that is allocated to the project conducts an independent review. The specific methodologies and activities employed during a smart contract audit may vary across companies, but these typically involve:

1) The *execution of tests* and their evaluation: Auditors assess the adequacy of the test suite to determine whether it provides sufficient coverage of the contract's

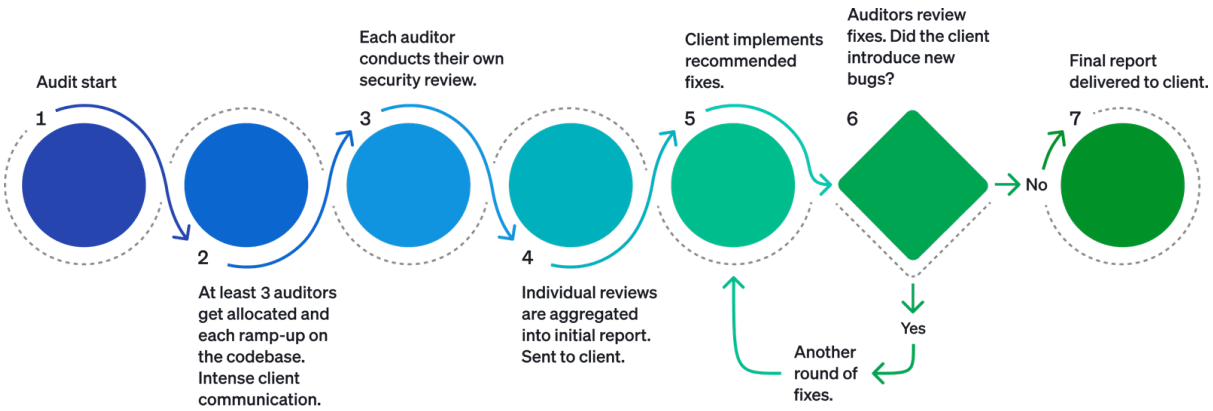


Figure 3.5: Audit process at Quantstamp

functionality. Code coverage analysis helps measure how much of the smart contract code is exercised by the existing test cases. While high coverage does not necessarily guarantee correctness, it serves as an indicator of test thoroughness and can highlight untested portions of the code that may harbor hidden bugs. If the coverage is low, auditors may recommend additional test cases to improve adequacy, ensuring that critical contract logic is adequately tested before deployment.

2) The execution of *automated tools*: Automated tools such as static analyzers, symbolic execution frameworks, and fuzzers are employed to uncover common bugs and vulnerabilities [6, 185]. *Slither* [61] is one of the most widely used tools, applying rule-based detection to identify reentrancy vulnerabilities, uninitialized storage variables, gas inefficiencies, and more. Instead, static analysis tools such as *Securify* [184] decompile smart contract bytecode to analyze its control flow and detect security flaws. *Oyente* [103] and *Manticore* [122] leverage symbolic execution to explore different execution paths and identify vulnerabilities. Fuzzing tools such as *Echidna* [70] randomly generate inputs to trigger unexpected behaviors in smart contracts, helping uncover edge cases that traditional testing may miss. Additionally, tools like *FSolidM* [112] enable security validation by modeling smart contract behavior as finite-state machines, allowing auditors to analyze contract properties using formal methods. Typically, these tools help auditors detect low-hanging bugs and security issues, but deeper, more sophisticated vulnerabilities might still require extensive manual inspection.

3) The thorough *inspection* of the smart contract source code: The manual review stage, introduced in Section 3.2.1, is a central activity of smart contract auditing, as it allows the reviewer to understand the logic and intricacies of the code in ways that automated tools cannot. Manual audits focus on contextual security risks, unexpected interactions between contract components, and the identification of bugs that might not be revealed through automated analysis. Code coverage insights obtained from earlier test evaluations can guide this process by helping auditors focus their attention on untested or poorly tested sections of the contract. If a part of the codebase remains unexecuted during testing, it may indicate that its behavior has not been adequately validated, prioritizing it for deeper inspection.

Once the audit team completes their review, they compile a comprehensive *report* detailing their findings. Issues are categorized based on severity, ranging from critical vulnerabilities that could result in loss of funds to minor inefficiencies. The report also

includes *recommendations* for addressing the identified issues, often suggesting specific code modifications. Following the initial audit, developers have the opportunity to implement the suggested fixes, after which a *follow-up* assessment may be conducted to verify that the issues have been properly resolved. The auditing firm may then issue a formal *certification*, which serves as an assurance to users and investors that the contract has undergone rigorous evaluation. Furthermore, a security audit is not a one-time event. Continuous monitoring and periodic re-audits are necessary, especially if the contract undergoes modifications through upgrades.

3.7 Regression Testing

The term *"regression"* denotes a return to an earlier, typically undesired, state. Regression testing is a fundamental process in software maintenance, ensuring that modifications (arising from bug fixes, feature additions or other types of change) do not introduce unintended defects in previously validated functionalities. In particular, it involves verifying that a modified program, denoted as P', not only *behaves correctly for newly introduced functionality* but also *preserves the correctness of pre-existing behavior* from an earlier version P. To achieve this, developers repeatedly execute the tests that were created earlier in the development process. Section 3.7.1 outlines the regression test selection problem and relative strategies, while Section 3.7.2 presents in more detail techniques for selecting modification-traversing tests.

3.7.1 Regression Test Selection

The necessity of regression testing arises in various software development scenarios, including feature additions, bug fixes, and performance optimizations. Given the extensive nature of test suites, running all tests from previous iterations is often impractical due to time and resource constraints. **Regression Test Selection (RTS)** strategies aim to efficiently identify a subset of tests that are sufficient to validate both new and retained functionalities in P'. The effectiveness of an RTS technique is typically evaluated based on its ability to reduce test suite execution time while maintaining fault detection capabilities.

- *Efficiency*: It relates to the reduction in the number of executed tests, impacting the overall computational overhead of RTS. Efficiency is critical in large-scale software projects where executing the full test suite is infeasible due to time and resource constraints.

- *Safety*: It ensures that an RTS technique never excludes tests that could detect faults in the modified program. Safety is crucial in mission-critical systems where missing a fault could lead to severe consequences.

Improving efficiency can sometimes come at the cost of safety. Similarly, strict safety requirements often result in selecting more tests than necessary, potentially reducing efficiency. Several strategies have been developed to address the RTS problem [111], each balancing efficiency and safety.

Test-All The most straightforward regression testing approach is to re-execute all (non-obsolete) tests from P on P'. This strategy guarantees maximum coverage but is often impractical due to high computational costs.

Random Test Selection A basic approach to reducing regression testing effort is to randomly select a subset of test cases from the existing suite. While this strategy lowers execution time, its effectiveness relies on the assumption that all tests are equally capable of detecting faults. This premise that rarely holds in practical software testing. Random selection presents a trade-off between efficiency and safety, often making it a suboptimal approach compared to more targeted RTS strategies.

Test Prioritization Instead of selecting a strict subset of tests, prioritization techniques reorder test execution based on predefined metrics such as fault detection likelihood, execution time, or code coverage. This enables testers to execute the most critical tests first, allowing rapid detection of regressions.

Selecting Modification-Traversing Tests A more sophisticated approach focuses on selecting tests that execute modified or potentially impacted code regions. This is achieved through static or dynamic analysis to determine dependencies between tests and program elements. Modification-traversing RTS techniques ensure that selected tests are directly relevant to the introduced changes. Safe RTS techniques, which ensure that all modifications are exercised by at least one selected test, provide strong fault detection guarantees while minimizing redundant test executions. A key advantage of modification-traversing techniques is that, in time-constrained scenarios, testers can execute a smaller but highly effective subset of regression tests. However, the sophistication of these techniques necessitates automation, as manual application to large commercial systems is impractical.

Test Minimization Test minimization techniques further refine selected test sets by identifying and removing redundant test cases. A test is considered redundant if another test in the selected set achieves the same verification objectives. While test minimization reduces execution time, it risks discarding tests that may reveal faults under specific conditions, making it a potentially unsafe strategy if not carefully validated.

3.7.2 Techniques for Selecting Modification-Traversing Tests

This Section provides an overview of RTS techniques that focus on selecting modification-traversing tests. These techniques all follow a similar high-level workflow, which is illustrated in Figure 3.6. Consider a program P tested against a suite T, where P evolves to a new version P'. Every such technique operates on the principle of identifying those tests in T that traverse *modified* or *impacted portions of the codebase*. Thus, instead of re-running all tests, only those traversing affected code paths are executed. To achieve this, RTS techniques rely on two primary data structures:

(1) *Test dependencies* establish the relationship between test in T and code in P;

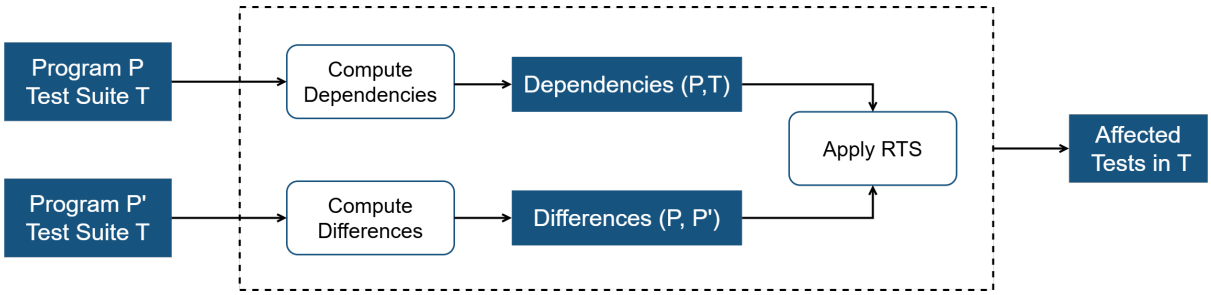


Figure 3.6: Regression Test Selection based on Modification-Traversing Tests

(2) *Program differences* capture changes between two program versions P and P'.

In the following, different approaches to RTS are described, considering: 1) *how test dependencies can be computed*, 2) *how program revision differences can be obtained*, 3) *how the selection itself is performed* starting from this data, and 4) how the chosen *granularity of computations* affects the whole process.

1) Granularity Considerations

The process of regression test selection uses code changes and tests dependencies to build a regression test suite for a new program revision. Computing test dependencies means keeping track of which parts of the program P the test suite T reaches. Computing program changes means understanding which parts of the program P changed since the previous revision P'. The granularity of computation determines the nature of these atomic parts. The selected granularity usually influences the specific technique used to collect the data and moreover, it can affect the safety and efficiency of the RTS. Granularity selection involves trade-offs between precision, computational overhead, and safety. Three primary levels of granularity are commonly employed.

Basic Block Level This is the most fine-grained approach. A basic block is a sequence of code statements with no branches in except to the entry and no branches out except at the exit. Any statement that does not affect control flow can appear inside a block. Using basic blocks granularity means that code changes are spotted and recorded in an extremely precise way. If a change occurs in a single line into a method, only those tests that traverse the modified line are selected. As a result, the regression test suite would be very accurate and minimized. But quality has a cost. While fine-grained techniques minimize test execution, they may impose more complex analyses and additional overheads. Indeed, due to the growing and relatively larger scale of modern software systems, many state-of-the-art RTS techniques (e.g., Ekstazi [67], STARTS [97], and FaultTracer [198]) use file or method granularity rather than considering statements or basic blocks, which are more expensive to collect.

Method Level This is a coarser-grained alternative where entire functions are treated as atomic units. If any statement in a function changes from P to P', all test methods that exercise the function will be considered affected, and included in the regression suite. Being less precise, method-level RTS can increase test execution time compared

to the basic-block level. An example of method-level RTS tool for Java is *FaultTracer* [198]. However, it is still fine-grained enough to impose complex analyses and additional overheads. Additionally, studies show that method-level granularity does not always result in significant benefits with respect to file-level approaches. Consider Ekstazi, a dynamic state-of-the-art RTS tool for Java [67]. In a study, the authors show that the use of coarse-grain dependencies provides a "sweet-spot" [68], and that it can provides better results than selecting tests at the method-level.

File (or Class) Level This is the most conservative approach, treating entire files as atomic units. Any change within a file (e.g., a Java class) triggers re-execution of all tests related to the file. Even though file-level approaches may over-select tests, they favor simplicity and safety and are more widely applied in real-world development. As already mentioned, one dynamic state-of-the-art RTS tool for Java called Ekstazi [67] showed the benefits of file-level RTS. Subsequently, an extensive study investigated the performance benefits of static RTS techniques and their safety [96]. The authors implemented two static RTS techniques, one class-level and one method-level, and compared several variants of these techniques, showing that the method-level static RTS technique performs rather poorly.

2) Computing Test Dependencies

The inputs to a traditional RTS technique are two software revisions (new and old), and the dependency information from the test runs on the old revision. The computation of test dependencies is central to RTS and can be approached dynamically or statically.

Dynamic Techniques Dynamic RTS techniques derive test dependencies from execution traces obtained by running test cases in T against the original program version P. These techniques often construct the control flow graph of P, where nodes represent basic blocks of code and edges define execution paths. While running tests, an execution trace records visited nodes, forming a test vector that maps each node to the set of tests that traverse it. Sometimes, dynamic RTS tools rely on existing coverage collection modules rather than building a Control Flow Graph (CFG) or implementing ad-hoc analyses.

Techniques that collect finer-granularity dependencies may be more precise, selecting fewer tests to be run, but can incur higher analysis overhead. *FaultTracer* [198] is a state-of-the art approach that collects dependencies dynamically by constructing an Extended Call Graph (ECG) at the method and field level. Unlike traditional call graphs that capture only method calls, ECGs also track field accesses (reads and writes), offering a more precise view of test dependencies. This is achieved through bytecode instrumentation, where FaultTracer injects monitoring code at runtime to trace method invocations and field accesses.

Instead, techniques that collect coarser-granularity test dependencies may be less precise but can have lower analysis overhead. A state-of-the-art, dynamic RTS tool for Java is *Ekstazi* [67]. Ekstazi tracks changes and dependencies at the granularity level of files. In particular, it determines dependencies by monitoring file accesses during test execution. It records which files are accessed by each test and stores their checksums

to detect changes. When the program is modified, Ekstazi compares the checksums of the dependent files with their previous versions to determine which tests need to be re-executed. Prior experiments with Ekstazi [68] showed that despite this coarse-grained selection of test dependencies, its RTS approach can still substantially save the end-to-end testing time (this includes the time to analyze changes, run selected tests, and update dependencies).

Although dynamic dependency collection is widely studied and used, it may not always be suitable for different reasons [96]. Indeed, dynamic test dependencies for large projects may be time consuming to collect, especially when they introduce code instrumentation. For programs with non-deterministic paths, dependencies collected dynamically may not cover all possible traces. Additionally, dynamic dependencies must be computed on the original program revision P. if changes in the source code divert the flow, old execution traces may become obsolete, leading to dynamic RTS being unsafe.

Static Techniques Static RTS techniques collect dependencies by analyzing the structure of tests in T and of the code in P, without executing test cases. They construct *dependency graphs* based on call relationships to approximate test dependencies conservatively. The underlying structure of a dependency graph is a directed graph where each node points to the node on which it depends. For example, consider the program in Listing 3.6 and its test classes in Listing 3.7. When using static analysis, the dependencies between tests and the program code are inferred from method calls and class hierarchies without considering actual runtime behavior. Depending on the granularity of computation, dependency graphs and dependencies can change, but this aspect will be discussed later. Here, a method-level scenario is assumed.

```
1  // Library code
2  class L {
3     void m1 () {}
4  }
5
6  // Source code
7  class C1 extends L {
8    void m1 () {
9      C2.m3();
10   }
11   void m2() {}
12 }
13
14 class C2 {
15    static void m3() {}
16 }
```

Listing 3.6: Program P

```
1  // Test code
2  class T1 {
3    void t1 () {
4     L l = new L();
5     l.m1() ;
6   }
7  }
8  class T2 {
9    void t2 () {
10    L l = new C1();
11    l.m1();
12   }
13 }
14 class T3 {
15   void t3 () {
16    C1 c = new C1();
17    c.m2();
18   }
19 }
```

Listing 3.7: Test Classes for P

The call graph shown in Figure 3.7 is a result of such an analysis for program 3.6. Intuitively, starting from each root (e.g., the main function), the call graph construction

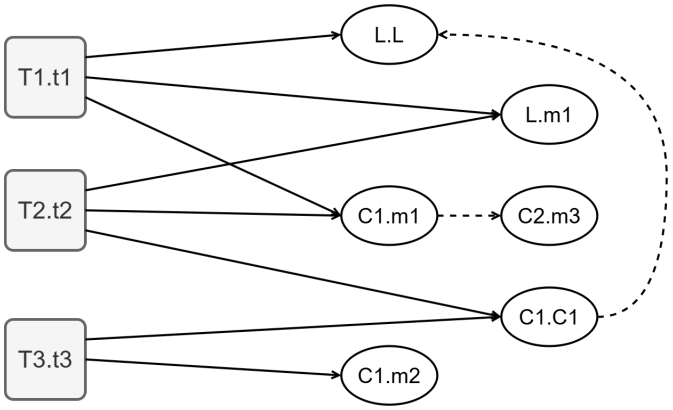


Figure 3.7: Call Graph for Program P

finds all functions that can be transitively invoked from the root function. The graph permits to store and retrieve test dependencies for regression test computation. However, the call graph considers all potential rather than actual execution paths. Thus, methods may be assumed reachable due to conservative approximations. Static analysis generally leads to safer regression test suites, as it is unlikely that dependencies will be missed, but it can also select more tests than necessary due to imprecise dependency inference. Static approaches are still beneficial when safety is a major concern with respect to computational and time savings.

STARTS [97] is an example of mature RTS plugin for Java that relies on program structure analysis rather than execution traces. Unlike Ekstazi, which dynamically tracks file dependencies during test execution, STARTS requires no code instrumentation or runtime information to find impacted tests. Instead, it uses only compile-time information. Specifically, it builds a dependency graph of program types and marks as impacted tests that can reach some changed type in the transitive closure of the dependency graph. A key advantage of STARTS is its ability to perform RTS without requiring execution traces, making it suitable for environments where runtime analysis is infeasible. The authors later evaluated several variants of file and method-level static RTS [96]. The results show that class-level static RTS is comparable to Ekstazi, but method-level static RTS is rather poor. The key message of the study is that static RTS at the class level shows promising results, and researchers should continue to improve it. Although static RTS can result in the selection of unnecessary tests, it could be more beneficial than dynamic RTS for systems with long-running tests, non-determinism, or very strict safety requirements.

3) Computing Program Differences

An RTS technique works by finding the dependencies of each test and selecting tests that depend on the changes. Thus, as illustrated in Figure 3.6, RTS also require a way to identify modifications between any two program versions P and P'. The technique used often depends on the chosen granularity of computations. Techniques that already compute program dependencies at a fine-grained level (e.g., the dependencies between test methods and program methods), often construct the CFG or the Abstract Syntax Tree (AST) for both P and P' and compare corresponding nodes. Differences within AST

or CFG nodes indicate modified code regions requiring regression testing. For example, FaultTracer [198], traverses the ASTs of both versions and compares methods and fields by their fully qualified name. Instead, techniques that operate at a more coarse level usually rely on more lightweight techniques. For example, STARTS [97] determines the types that changed by computing the checksum of each compiled class file and comparing it with the one computed in the prior run. Ekstazi [67], which also works at the file level, stores the name and checksum of the program classes used by each test class during execution.

4) Regression Test Selection Process

The final step in RTS involves combining test dependencies and program differences to determine the regression test set. This step is strictly related to the techniques used for the computation of test dependencies and programs differences. The relative data structures, combined with a selection algorithm, will produce the regression tests subset. A general selection algorithm typically starts by traversing the ASTs or CFGs of the original and modified programs (or by simply analyzing the checksums) to identify nodes with structural changes. Then, it selects all tests associated with affected nodes (or files, in the case of checksum-based approaches). This ensures that the chosen test subset effectively verifies modifications while maintaining confidence in unchanged components.

3.8 Capture-Replay Testing

Capture-Replay testing (also known as Record-Replay testing) is a dynamic testing technique employed primarily during *System* or *Integration Testing*. This approach revolves around the idea of recording real user interactions with a system under test (SUT) and subsequently replaying them to validate software behavior. The value of capture-replay lies in its ability to *automate realistic usage scenarios* that might otherwise require extensive manual effort, and *pinpoint regressions or discrepancies* in the SUT's behavior when compared to the original recorded interactions. Capture-replay testing is very flexible and particularly valuable when reliability and realism of test interactions matter. Key applications of this technique include:

- *Regression Testing*: Testers can replay captured interactions to identify when a code change introduces undesired behavior or breaks existing features;
- *Performance Testing*: Testers can use realistic production traffic to validate the performance of the system under conditions that mimic real-world usage;
- *Production Crash Reproduction*: Captured interactions can help replicate the conditions that led to a failure in production and diagnose the issue;
- *Debugging*: Replay logs show how the system reacts to recorded events, making it easier to isolate problematic code segments and locate bugs.

This technique has been applied in a variety of software domains. In the context of mobile applications, for example, capture-replay tools automate what would otherwise

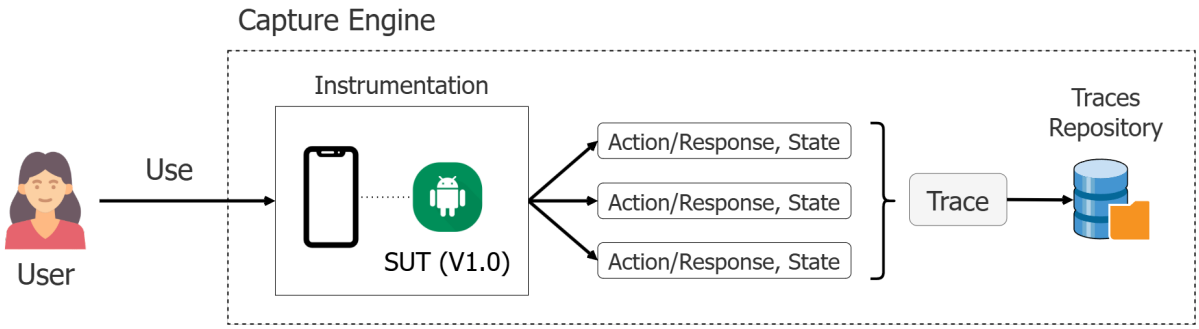


Figure 3.8: Capture Phase in the Capture-Replay Testing Process

be a tedious, manual process of verifying app consistency on different devices and configurations [28, 143, 62]. The most typical application of replay tests, however, is to detect unexpected system behavior that may have been introduced after system changes or updates [9, 73]. By recording online traffic during normal operations, replay testing can build a robust suite of regression tests without requiring a deep knowledge of the system. The capture-replay testing process comprises two main phases:

1) Capture: During the *Capture* phase (illustrated in Figure 3.8), the tool's capture engine records user interactions with the System Under Test (SUT). In this example, the SUT is an Android application, but the process is similar in other software domains. The captured interaction typically includes detailed information about user inputs, system responses, and their context. Examples include keystrokes, HTTP requests and responses in web applications [5, 102], or user gestures and touch events in mobile applications [28, 143, 62]. To ensure replay consistency, the capture process may also record relevant system state at key points. This may include UI states, database snapshots, session information, or any other contextual data needed to restore the SUT to its exact condition at the time of capture. The level of data granularity varies depending on the requirements, ranging from high-level user actions to low-level system calls. In some cases, capture-replay solutions require *instrumentation*, where modifications to the application code or environment enable event logging. This may be necessary to obtain detailed interactions but could introduce overhead and degrade user experience. Alternatively, some tools avoid instrumentation by capturing user interactions at the OS level or leveraging system logs. Finally, captured actions, responses, and state snapshots are structured into *traces* and stored in a *repository*.

2) Replay: During the *Replay* phase (illustrated in Figure 3.9), the testing tool retrieves a previously captured trace and replays it step by step against a potentially updated version of the system under test (e.g., a newer application version). One of the challenges in replay testing is managing the potentially large volume of recorded traces. In many cases, the amount of data captured can be overwhelming, with a significant portion of the interactions being redundant or less critical for the testing objectives. Therefore, selection strategies are often employed to choose the most relevant interactions to replay. Once a trace is selected, the system state is often reset to match the conditions recorded

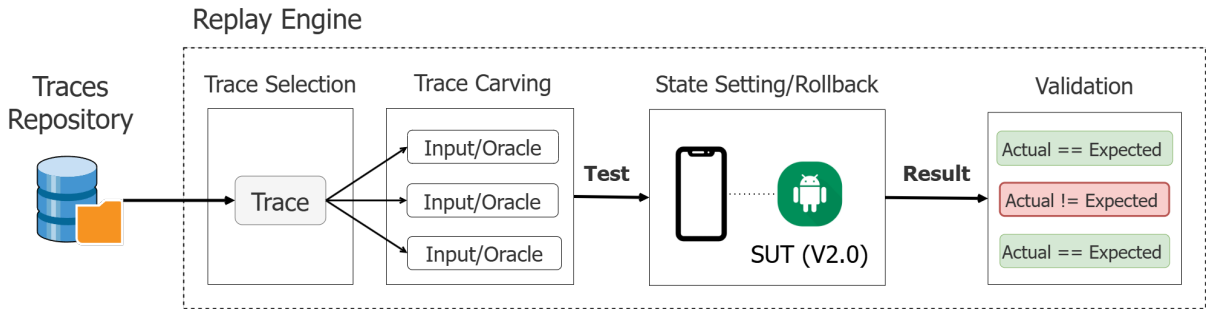


Figure 3.9: Replay Phase in the Capture-Replay Testing Process

during the capture phase to ensure consistency. This state reset can involve restoring databases, resetting UI elements, or reapplying session contexts to avoid discrepancies. Once the state is restored, the replay engine injects the captured interactions into the SUT, simulating the same sequence of user actions recorded during capture. For example, if a captured test involved filling out and submitting a form, the tool will automatically input the same data into the form, submit it, and observe the resulting system behavior. The outputs generated by the SUT during replay are compared against the expected outputs stored in the trace. A *validation* module identifies any mismatches between expected and actual behavior, flagging them as potential regressions or inconsistencies. The final results are logged, helping developers diagnose and resolve issues introduced by system modifications.

CHAPTER 4

CHALLENGES IN SMART CONTRACT QUALITY ASSURANCE

Smart Contracts are the most critical component of any dApp because they define the business logic that is executed on a blockchain network. Due to their immutability and to the financial implications associated with their operations, smart contracts demand higher reliability guarantees when compared to traditional software systems, and thus require special attention from the testing community. Although smart contract verification and validation constitute a significant portion of blockchain research, testing best practices and tools are not yet mature. As a result, it is complex for smart contract developers to ensure the deployment of reliable code.

This Chapter discusses the challenges that this thesis aims to address, expanding on the Research Objectives (ROs) defined in Section 1.2. In particular, Section 4.1 introduces the smart contract development life-cycle, describing its key phases and how they differ from traditional software. The research objectives were defined in relation to three main phases in this life-cycle: *testing* (RO1), *auditing* (RO2) and *maintenance* (RO3). For the *testing* phase, Section 4.2 provides an overview of test adequacy assessment tools and approaches for smart contracts, identifying open challenges and areas for improvement. Section 4.3 describes smart contract *auditing* in more detail, and discusses challenges in the adoption of mutation testing within this process. For the *maintenance* phase, Section 4.4 highlights the key challenges in the validation of upgradeable smart contracts.

4.1 Quality Assurance in Smart Contract Development

The development life-cycle of a smart contract is different from that of conventional software, mostly due to the immutable nature of blockchain technology. Unlike traditional software, where updates and patches can be applied post-release, smart contract cannot be directly modified after deployment to the blockchain. Although mechanisms exist to

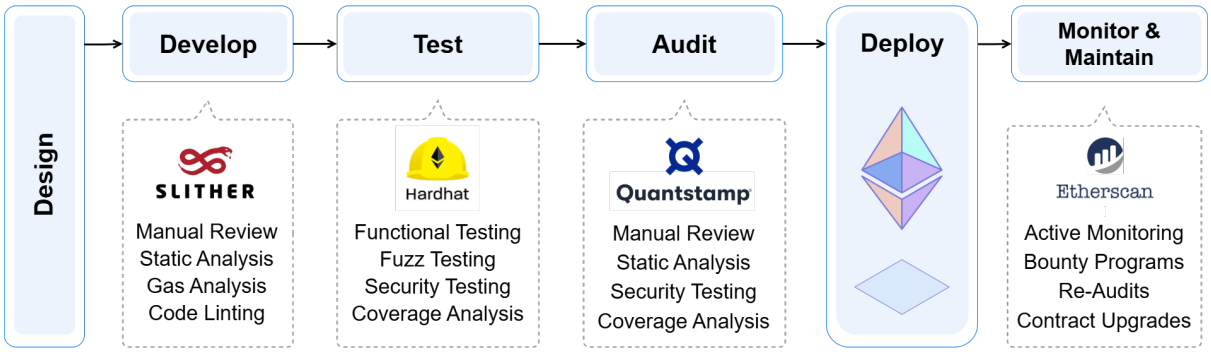


Figure 4.1: Quality Assurance Activities in the Smart Contract Development Life-Cycle

allow for upgrades, they must be carefully planned at design-time, often through specific architectural patterns, which introduces additional layers of complexity and security considerations. These constraints, combined with the potential financial risks associated with blockchain applications, led to a development approach that heavily **prioritizes** rigorous **pre-release testing** and extensive **external audits**. Figure 4.1 illustrates the main phases in the smart contract development life-cycle. For each phase, from development to maintenance, it shows some of the key tools and quality assurance activities performed within them. Note that these are purely illustrative, as each development team might rely on specific frameworks (e.g., HardHat for testing), tools (e.g., Slither for static analysis), and external companies (e.g., Quantstamp for auditing).

1 - Requirement Gathering and Design The foundation of quality assurance in smart contract development is established during the requirement gathering and design phase. From a conceptual perspective, this does not differ much from that of a traditional software. The design phase involves defining the contract's business logic and eventual interactions with other system components, such as events, functions, and the flow of funds or data. However, in smart contract development, particular attention must be given to the constraints imposed by the blockchain, including gas costs, security risks, and the integration with off-chain systems (e.g., price oracles). In the early years of Ethereum, the lack of well-established best practices and standards made building high-quality Smart Contract more challenging. Since then, a lot of effort from academia and industry went into blockchain-oriented software engineering [51, 185], providing collections of design patterns and best design practices.

2 - Development The development phase involves translating the contract's logic into code using Solidity or Vyper. In general, in smart contract development there is a heightened focus on secure coding practices and code optimization to ensure gas efficiency. Like for the design phase, many development patterns and best practices have been proposed and refined through the years. For example, Kannengießer et al. [90] identified 29 challenges in smart contract programming (e.g., code visibility and upgradeability) and proposed 20 design patterns in collaboration with smart contract developers. Marchesi et al. [109] collected a list of security patterns for DApps and provided security assessment checklists for the development of smart contracts. Additionally, developers typically

perform manual code reviews and apply static analysis tools (e.g., *Slither* [61] to catch bugs and inefficient code patterns as early as possible.

3 - Testing Performing extensive testing is the first and most important line of defense for delivering reliable code, especially given the restrictions introduced by post-deployment immutability. Typically, developers write functional tests for individual functions and to evaluate interactions between different components, including other smart contracts, external data sources (oracles), and user interfaces. Tools like Hardhat allow developers to mimic the Ethereum environment and test the contract's behavior in realistic scenarios, without incurring transaction fees. Advanced testing techniques such as fuzz testing (e.g.: Echidna [70]) may also be used to examine contract behavior under diverse conditions. At the same time, developers may rely on coverage analysis to evaluate and improve the quality of their test suites.

4 - External Audit and Certification After internal testing, the project team may request an *audit*. As introduced in Section 3.6, an audit is an independent, third-party assessment of the smart contract's security and overall functionality. Smart contract auditors are experts who focus on blockchain-specific vulnerabilities, such as reentrancy and front-running attacks, gas optimization issues and other threats. When the audit is concluded, they write a formal report detailing any identified bugs or vulnerabilities, their severity, and the corrective actions that the team must implement. In general, external audits serve as a seal of approval that demonstrates the team's commitment to high-quality standards. Clearly, the efficacy of the auditing process depends both on the expertise of the auditors themselves, and on the availability of innovative testing methods and tools.

5 - Deployment Once the smart contract has passed all testing phases it is deployed to the Ethereum blockchain, where it becomes publicly accessible and operational. Unlike traditional software, deploying a smart contract requires the payment of a fee, and it permanently associates the contract code with a public address. After deployment, the development team typically publishes the smart contract's source code (and other artifacts) on platforms like Etherscan to boost transparency and community trust. When users interact with a smart contract, they are effectively signing a digital transaction that could have irreversible consequences. Developers that publish the source code allow users to verify that the on-chain bytecode truly reflects the intended functionality. Releasing any report or certification obtained from auditing companies further demonstrate the team's commitment to high-quality standards.

6 - Monitoring and Maintenance Although deployed smart contracts are immutable, they still require maintenance like any other software, especially if bugs are discovered within their implementation. Unlike conventional programs, however, smart contracts require *specialized mechanisms* to *accommodate changes*. In traditional (i.e., non-upgradeable) smart contracts, changing the code in any way means redeploying a new version of the old smart contract at a different address on the blockchain. Upgradeable Smart Contracts, such as those implementing the proxy pattern (discussed in

Section 2.3.4), allow developers to update contract logic without altering its state, thus enabling continuous improvement and patching. Thus, during this phase, quality assurance practices focus heavily on *monitoring* contract activity to detect anomalies, such as unexpected transactions or malicious behavior. Dedicated platforms (e.g., *OpenZeppelin, Etherscan)* and analysis tools enable real-time monitoring of smart contracts, and transactions analysis to identify potential threats before they escalate.

4.2 Smart Contract Testing

This Section discusses the challenges in smart contract testing that the thesis aims to address. Testing is a broad topic with numerous open research directions. As stated in RO1, the thesis focuses specifically on the lack of **advanced test adequacy assessment techniques** for Solidity smart contracts, and their practical integration within their development life-cycle. This limitation affects the ability of practitioners to thoroughly evaluate the quality of their test suites.

4.2.1 Test Adequacy Assessment

One common metric for assessing test adequacy is code coverage analysis, which measures the extent to which a program's code is executed during testing. Tools for coverage analysis, such as *solidity-coverage* [150], are available for Solidity and are relatively simple to use and interpret. However, as discussed in Section 3.4, high code coverage does not guarantee that the test cases are meaningful. In the context of smart contracts, where financial stakes and security risks are high, relying solely on coverage analysis can result in a false sense of confidence in the test suite.

Mutation testing offers a more advanced approach to test adequacy assessment. In the literature, it is possible to find some proposals that implement a mutation testing approach for Solidity: *MuSC* [192], *ContractMut* [75], *Vertigo* [76], *Universal Mutator* [3], *RegularMutator* [84] and *Deviant* [40]. There is also a GitHub repository with a related tool, *Eth-mutants* [60], which however only implements a generic mutation operator, and is thus not considered.

Even so, the use of mutation testing in Solidity is still very limited, and the review of existing work reveals several gaps and limitations that should be addressed. Overall, the absence of a standardized approach to mutation testing has led to a *fragmented ecosystem* where tools are narrowly focused or only partially applicable. Existing solutions are discussed in the following, focusing on the implemented *mutation operators*, *tool compatibility*, and other *relevant features*.

Mutation Operators Existing tools adopt largely different sets of mutation operators. Wu et al. propose *MuSC* [192], the first mutation testing framework for Solidity. *MuSC* generates the AST of the target contracts and generates mutants using nine JavaScript-oriented and fifteen Solidity-specific mutation operators. These were designed from the Solidity documentation and by looking over smart contract related issues on GitHub. Since the proposed set is moderately compact it limits the number of generated mutants.

However, it is also limited in scope, lacking several operators that can introduce severe faults (e.g., those targeting ether transfer mechanisms or function modifiers).

The *ContractMut* tool proposed by Hartel & Schumi [75] implements general mutation operators derived from the minimum standard Mothra set, as well as four Solidity-specific mutation operators. To limit the number of generated mutants, the authors apply *selective mutation* with a focus on mistakes concerning access control.

The *Vertigo* tool proposed by Honig et al. [76] introduces useful cost-reduction features, but it only implements two Solidity specific operators (targeting function modifiers and increment statements) and four general operators.

The approach proposed by Andesta et al. [3] is based on the study of known bugs in Solidity smart contracts. The core idea is that designing operators capable of recreating real-world bugs can be an effective way of selecting good test cases. This led to the definition of 10 classes of operators that cover many types of smart contract vulnerabilities. These operators were included in *Universal Mutator* [72], a popular regexp-based mutation testing tool that can be adapted to different programming languages.

Similarly, *RegularMutator* [84] uses regular expression to generate Solidity mutants, instead of relying on a language parser. However, the validation shows high numbers of compilation error (≈ 84 of the generated mutants), making the approach highly unreliable.

Chapman [40] proposes *Deviant*, an open-source mutation testing tool that aims to help developers in delivering higher quality code. In this case, a broader set of 62 Solidity-specific mutation operators was designed according to the Solidity fault model.

Compatibility with Ethereum Testing Frameworks Ethereum's smart contract ecosystem offers a variety of testing frameworks for writing, deploying, and testing smart contracts. Each of these frameworks, including Truffle [45], Hardhat [124], Brownie [57], and Foundry [63], support different features, various plugins for testing smart contracts, and different languages for writing test cases. This diversity allows developers to choose solutions best suited to their needs, but it also introduces challenges in maintaining compatibility across different testing approaches. Furthermore, given the rapid pace of innovation in tooling, plugin adaptability is essential to ensure long-term compatibility with emerging frameworks. This is also the case for mutation testing.

Table 4.1 highlights a lack of standardization among existing mutation testing tools for Solidity. Most tools are designed to integrate exclusively with *Truffle*, a framework that is no longer actively maintained. *Vertigo* supports the evaluation of tests for *Hardhat*, a more modern and widely adopted framework, but it only implements a couple of mutation operators for Solidity. The *Universal Mutator* tool implements a more comprehensive set of Solidity operators, but the tool itself only focuses on mutant generation, delegating the actual testing process to the user. Instead, no tools currently support the evaluation of *Brownie* and *Foundry* tests. This fragmentation in itself limits the broader adoption of mutation testing in the Solidity ecosystem.

Other Features Some tools provide additional features that aim to make the mutation testing process less costly and the results more reliable. For example, *ContractMut* is the only tool that identifies and discards equivalent and redundant mutants using the trivial compiler equivalence, or TCE [133]. This work also advances the state of the art by introducing a novel killing condition for the mutants based on the gas consumption of

Table 4.1: Comparison of Mutation Testing Tools

Tool	**Compatibility**				**Other Features**		
	Truffle	HardHat	Brownie	Foundry	TCE	Mutant Sampling	Parallel Execution
[75] `ContractMut`	✓	x	x	x	✓	✓	x
[40] `Deviant`	✓	x	x	x	x	x	x
[192] `MuSC`	✓	x	x	x	x	x	x
[84] `RegularMutator`	✓	x	x	x	x	x	x
[3] `UniversalMutator`	x	x	x	x	x	x	x
[76] `Vertigo`	✓	✓	x	x	x	✓	✓

transactions. Vertigo applies *random mutant sampling* after the mutants are generated to reduce the process runtime. Other tools directly prevent the generation of mutants by only implementing specific operators in their set. Both selective mutation and sampling permit to save time, but if they are too drastic or only permit the generation of a few mutant categories (e.g., access control-related ones) it is not possible to derive a high-quality test suite. *Vertigo* is also the only tool that implements (local) parallel mutant execution to speed up the testing process. In any case, focus on mechanisms and techniques for boosting the application of mutation testing remains largely limited.

Overall, research on mutation testing for Solidity primarily focuses on defining sets of mutation operators and detailing their technical implementation. Existing tools often cater to narrowly scoped research prototypes and lack the robust features required by developers. Different tools introduce various sets of mutation operators, often targeting specific types of smart contract faults (e.g., access control vulnerabilities) or known bug patterns. While these selective approaches reduce the number of generated mutants, they often compromise the thoroughness of the testing process. Additionally, some early tools have not been updated and still rely on outdated mutation operators that are incompatible with newer Solidity versions. From a practical standpoint, many implementations only support the now-obsolete *Truffle* framework or focus solely on generating mutants without addressing the execution of tests. To establish mutation testing as a valuable technique in the blockchain domain, it is important to make the process comprehensive and practical for developers.

4.2.2 Continuous Mutation Testing

Despite its clear benefits in identifying gaps within a test suite, mutation testing remains an underused technique among Ethereum smart contract developers. Although existing research has contributed to the foundational understanding of Solidity mutants, it has not considered its practical integration into the typical smart contract development life cycle. In modern software engineering, developers expect testing tools to integrate easily with

existing workflows, supporting incremental updates, rapid feedback loops, and regression testing strategies. However, most existing mutation testing studies and tools envision the technique as a standalone process that is run only when the test suite has reached a certain level of maturity or immediately prior to deployment. This may work for a one-time snapshot of test quality, but it cannot support incremental improvements or maintenance. Specifically, this perspective creates two key problems:

1) Running Continuous Mutation Testing during development The first issue is that evaluating the test suite hand-in-hand with smart contract evolution becomes impractical. Even relatively small smart contract projects can take hours to complete a full mutation testing run. This only gets worse as the size and complexity of a codebase expands. Under these conditions, developers who wish to use mutation testing during day-to-day development are essentially forced to postpone the testing until right before deployment. This also means developers must address all surviving mutants at once, which can be overwhelming and discourage the use of the technique.

2) Running Mutation Testing on new Smart Contract releases Second, Ethereum smart contracts are often subject to stringent deployment constraints that make post-deployment updates challenging. Once a contract is deployed on-chain, modifying it typically involves intricate upgrade processes (e.g., via proxies). Each new release must be extensively validated, but running a mutation testing campaign from scratch can be unnecessarily time-consuming and delay release. Considering the importance of timely security updates and bug fixes, this is not ideal, and developers might just decide to skip the re-evaluation of test cases.

The overall effect is that mutation testing, despite its potential, is *often seen as an experimental method* rather than a standard part of development. Addressing these challenges is critical for encouraging broader adoption of mutation testing in the Ethereum ecosystem. In particular, mutation testing frameworks should focus on the analysis of impacted smart contracts and test code, providing faster feedback on how incremental modifications impact test quality.

4.3 Smart Contract Auditing

Research Objective 2 revolves around the smart contract auditing phase. The process of auditing, which was introduced in Section 3.6, is an independent and systematic evaluation of a smart contract that aims to increase its security, reliability, and overall correctness. Mutation testing is a well-established technique for evaluating the effectiveness of test suites. However, its adoption in smart contract auditing remains limited. Auditors typically analyze code coverage as part of their workflow to provide feedback to clients on the quality of their test cases. In theory, mutation testing could enhance this process by offering deeper insights into test quality and enabling auditors to deliver more detailed feedback. Despite its potential, mutation testing is rarely integrated in auditing processes. In fact, it is often viewed as a high-cost activity with limited immediate benefits.

This section describes smart contract auditing practices more in-depth and discusses key challenges in the *adoption of mutation testing* in this domain.

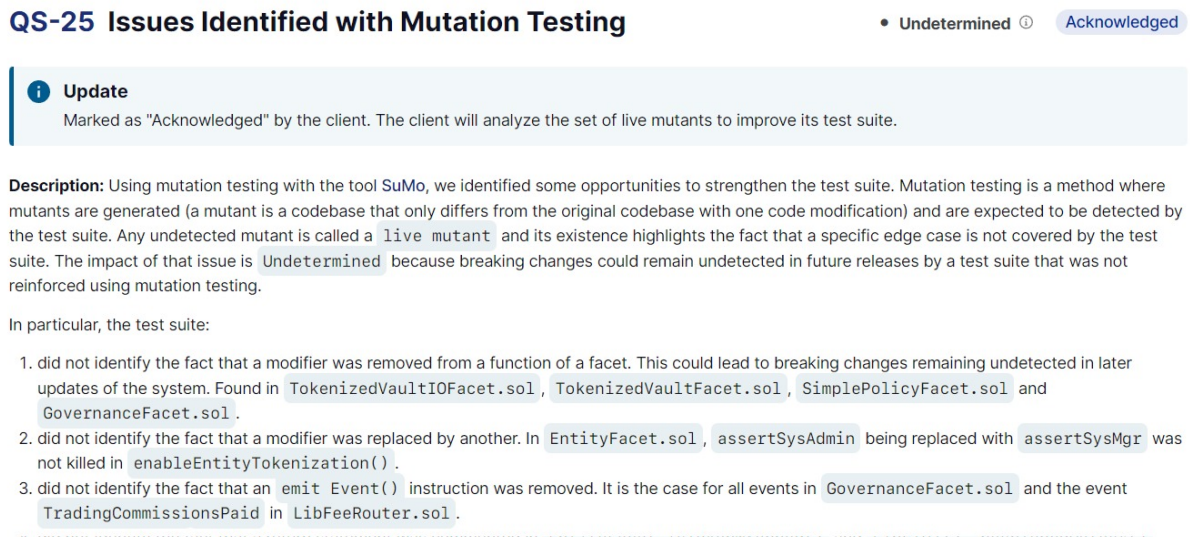
4.3.1 Adoption of Mutation Testing

For mutation testing to gain traction in smart contract auditing, this thesis identified several key challenges and corresponding research directions that need further exploration.

1) Surfacing Live Mutants During Manual Code Inspection: First, auditors generally *do not engage directly in test improvement.* Their primary responsibility is to review the codebase, identifying vulnerabilities, flaws, and areas for improvement. Mutation testing is exclusively seen as a technique for improving test quality, which is often a secondary concern for auditors. However, previous studies in the industry [139] suggest that mutants can occasionally assist in code inspection by uncovering issues in the code under test in addition to optimizing the test suite. This raises the question of whether mutation analysis in Solidity might similarly aid auditors in their manual code reviews. Specifically, mutation testing might provide insights into the behavior of the code and highlight potential issues that would otherwise be overlooked. For example, identifying specific mutation operators that generate particularly revealing mutants could guide auditors toward problematic sections of the code. In such a case, live mutants could be surfaced directly during the code review process, enhancing the effectiveness of manual inspections without requiring significant overhead.

2) Deriving Test Cases from Live Mutants For auditors, analyzing code coverage and providing high-level feedback on test quality to clients is relatively simple and straightforward. However, the insights derived from coverage analysis alone are often limited. Auditors can inform clients whether the test suite covers both happy and unhappy paths, confirm that all test cases pass, and assess whether the achieved coverage levels are satisfactory. Yet, if coverage levels are low, clients may still lack clear, actionable guidance on how to improve their test suite. Figure 4.2 presents an audit report produced by Quantstamp after running SuMo (introduced in Chapter 6) on a project under review. As shown in the report, the analysis goes beyond simple coverage metrics, providing more specific insights based on undetected Solidity mutants. Rather than just identifying areas of code that need more testing, auditors were able to pinpoint particular weaknesses, such as insufficient testing for function modifiers, event emissions, and other key Solidity features. However, manually analyzing hundreds of live mutants and understanding their implications is *time-intensive and mentally demanding.* This challenge could be in part addressed with the automatic derivation of test cases from live mutants. This approach would reduce the cognitive burden on auditors and allow them offer precise test suggestions to clients.

3) Establishing an End-to-End Mutation Testing Framework: The *lack of standardization* in tooling (see Section 4.2.1) means that setting up and running mutation testing is often complex, especially given the diversity of testing environments used across Solidity projects. This also means that auditors would need to run different sets of mutation operators for different projects, making it impossible to establish a common baseline for evaluation. At the same time, the process of running mutation testing is *time-consuming*, which delays the delivery of results and their analysis. This makes mu-



QS-25 Issues Identified with Mutation Testing

• Undetermined ⓘ Acknowledged

Update

Marked as "Acknowledged" by the client. The client will analyze the set of live mutants to improve its test suite.

Description: Using mutation testing with the tool SuMo, we identified some opportunities to strengthen the test suite. Mutation testing is a method where mutants are generated (a mutant is a codebase that only differs from the original codebase with one code modification) and are expected to be detected by the test suite. Any undetected mutant is called a `live mutant` and its existence highlights the fact that a specific edge case is not covered by the test suite. The impact of that issue is `Undetermined` because breaking changes could remain undetected in future releases by a test suite that was not reinforced using mutation testing.

In particular, the test suite:

1. did not identify the fact that a modifier was removed from a function of a facet. This could lead to breaking changes remaining undetected in later updates of the system. Found in `TokenizedVaultIOFacet.sol`, `TokenizedVaultFacet.sol`, `SimplePolicyFacet.sol` and `GovernanceFacet.sol`.
2. did not identify the fact that a modifier was replaced by another. In `EntityFacet.sol`, `assertSysAdmin` being replaced with `assertSysMgr` was not killed in `enableEntityTokenization()`.
3. did not identify the fact that an `emit Event()` instruction was removed. It is the case for all events in `GovernanceFacet.sol` and the event `TradingCommissionsPaid` in `LibFeeRouter.sol`.

Figure 4.2: Example of Audit Report listing Issues identified with Mutation Testing

tation testing less practical for auditors, who often operate under tight schedules. To be truly effective in auditing workflows, mutation testing results must be made available to auditors as early as possible. This would give them enough time to surface live mutants during code reviews, or to derive test case from mutants using other approaches. Therefore, establishing a standardized end-to-end mutation testing framework is a key area for exploration. The framework should support a wide range of projects while providing a uniform set of mutation operators. Beyond this, the framework should also include infrastructure enhancements to improve usability and streamline the mutation testing process. For example, a service designed to speed up test execution and an extension to visualize results could significantly improve the user experience.

4.4 Smart Contract Maintenance

Research Objective 3 revolves around the smart contract maintenance phase and its inherent challenges. A key issue in maintaining smart contracts stems from the lack of dedicated testing frameworks for validating upgrades. Among the existing upgradeability mechanisms, the proxy pattern has gained significant interest due to its effectiveness and simple design. Developers primarily use upgrades to improve contract usability, with the addition of functionality and updates being the most dominant intentions [101]. Currently, proxy-based projects are widespread on the Ethereum main chain, with 1.4 million proxy contracts used in more than 8k upgradeable systems [149]. Even Etherscan[1], the largest analysis and statistics platform for Ethereum, has added automated verification for proxy contracts, making it more intuitive for users to know which implementation contract they are interacting with.

Despite their flexibility, the usage of USCs can introduce risks. While upgrade patterns are now mature, well-documented, and embedded in many production dApps, the actual release of new contract versions remains infrequent [142]. This hesitation is largely

[1]Etherscan's Proxy Contract Checker: `https://etherscan.io/proxyContractChecker`

Main Objective	Role of Transactions
1) Active Monitoring and Attack Detection	- `DAppGuard` [46]: Actively monitors transactions to detect attacks and avoid security incidents.
2) Smart Contract Security Analysis	- `ContractVis` [74]: Replays transactions to evaluate contract dependencies and identify transparency issues. - `EthScope` [194]: Locates suspicious transactions and re-executes them to locate malicious smart contracts.
3) Forensic Analysis	- `TxSpector` [202]: Replays and performs forensic analysis on transaction to detect attack patterns. - `EthScope` [194]: Replays and analyzes transactions to reconstruct attack scenarios.
4) Contract Testing and Input Generation	- `SmartGift` [205]: Generates test inputs by learning from transaction records and extracts realistic parameter values.

Table 4.2: Usage of Blockchain Transactions in Quality Assurance

due to the risks associated with modifying smart contract logic, where even a seemingly small update could introduce ABI-breaking changes or disrupt underlying logic [101], resulting in failures in dependent systems. Subtle inconsistencies can lead to unintended consequences for dApps, off-chain services, and user interactions, making robust validation mechanisms an essential requirement for upgradeable contracts.

4.4.1 Validation of Smart Contract Upgrades

Validating smart contract upgrades, particularly in proxy-based architectures, presents a unique set of challenges. Even minor modifications can introduce regressions and break previously working contract functionalities. Additionally, smart contracts are rarely standalone. Other contracts, off-chain software, and external services often rely on their execution. Thus, changes that impact a contract's storage or function outputs may have cascading effects on dependent systems. A compatibility issue arises when an upgrade disrupts how external systems consume contract data or execute transactions.

While traditional testing approaches, such as unit testing and fuzzing, can identify localized issues, they may fail to assess the broader impact of an upgrade on real-world interactions. As introduced in Section 3.8, capture-replay testing allows developers to replay past interactions with a system and observe whether the new implementation maintains expected behaviors. Thus, by replaying historical transactions, developers can identify regressions or discrepancies before deployment. This approach can help ensure that new contract versions behave consistently under real-world usage and can help detect issues before they reach production.

However, there are currently no dedicated frameworks that systematically implement capture-replay testing for upgrade validation in proxy-based smart contract systems. As shown in Table 4.2, existing tools do leverage historical transactions, but they are designed for different purposes, including: *1) active monitoring and attack detection*, *2) smart contract security analysis*, *3) forensic analysis* and *4) contract testing and input generation*. For instance, DappGuard [46] intercepts real-time transactions to detect potential attacks. Other tools use historical transactions for analysis and testing. Con-

tractVis [74] generates test scripts from historical transactions and executes them to find opacity issues in smart contracts, while SmartGift [205] learns from past transactions to generate realistic test inputs, improving functional coverage. TxSpector [202] and EthScope [194] both provide transaction-level security analysis, replaying past transactions to detect attacks (e.g., reentrancy and unchecked calls) or to identify malicious smart contracts. Although transaction replay has somehow been considered, no tools explicitly addresses the challenge of ensuring that an upgraded contract maintains consistency with prior behavior. Thus, despite these advancements, there is a lack of a capture-replay framework tailored for upgrade validation. Developers are left with fragmented approaches, increasing the risk of deploying flawed upgrades that may only reveal their failures once they are live on the blockchain.

CHAPTER 5

SUMMARY OF CONTRIBUTIONS

This Chapter summarizes the main contributions of the thesis in relation to the three main Research Objectives presented in Section 1.2, and the specific challenges detailed in Chapter 4. To contextualize these contributions, Figure 5.1 illustrates the key frameworks proposed in the thesis and their positioning within the smart contract development life cycle. Each contribution addresses specific aspects of smart contract quality assurance, aligning with the broader goal of improving reliability and automation in the verification and validation of smart contracts. Section 5.1 provides a detailed discussion of the thesis's contributions to smart contract quality assurance, explaining how the proposed methods tackle the identified challenges. In addition, Section 5.2 highlights other relevant contributions to software testing that, while significant, are not explored in further detail within this thesis.

5.1 Smart Contract Quality Assurance

The high-level Research Objectives of the thesis, as described in Section 1.2, are: RO1: Supporting Test Adequacy Assessment for Smart Contracts, RO2: Supporting Mutation Testing in Smart Contract Auditing, and RO3: Supporting Smart Contract Maintenance

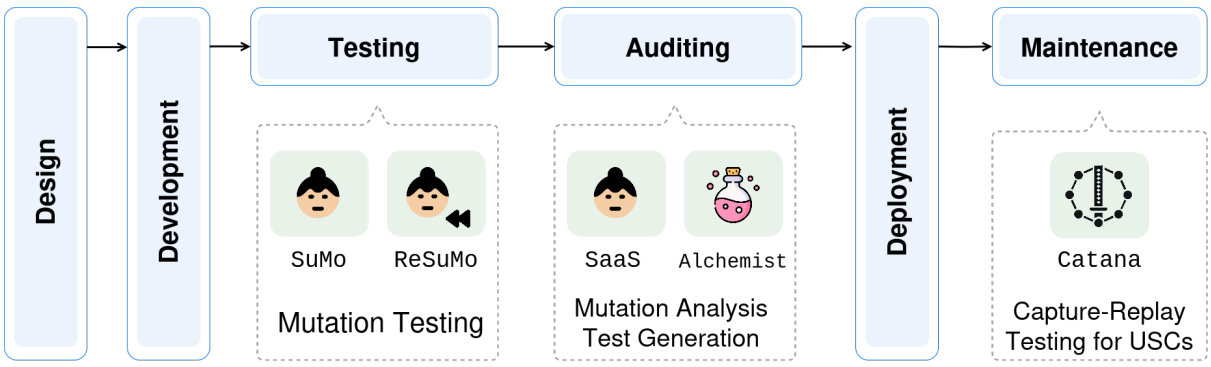


Figure 5.1: Contributions of the Thesis to the Smart Contract Development Life-Cycle

Practices. The following Sections present the main contributions of the thesis in relation to each RO.

RO1: Supporting Test Adequacy Assessment for Smart Contracts

Ensuring the reliability of smart contracts requires rigorous and effective testing methodologies. However, as identified in this thesis, existing testing approaches often lack mechanisms for assessing test adequacy, leaving developers uncertain about the effectiveness of their test suites. To address this gap, this thesis introduces novel techniques and tools aimed at improving smart contract testing, particularly through mutation testing. The contributions in this area focus on two key aspects: (1) advancing mutation-based test adequacy assessment for Solidity smart contracts and (2) enabling efficient regression mutation testing for continuous adequacy evaluation.

1) Test Adequacy Assessment for Solidity Smart Contracts. This thesis proposes SuMo, a novel mutation testing approach and tool for Solidity smart contracts [19, 22]. SuMo offers a comprehensive set of mutation operators that simulate Solidity-specific faults, thus enabling the derivation of higher-quality test suites. SuMo is available as an `npm` module compatible with existing Ethereum testing frameworks, and is intended for both practitioners and researchers interested in mutation testing. Our empirical evaluation of SuMo revealed significant gaps in open-source test suites, particularly around Solidity-specific programming features, and led to the detection of a real bug in one of the studied contracts. These findings show that SuMo can be a valuable addition to the developer's toolkit and contribute to the release of more reliable smart contract code.

2) Regression Mutation Testing for Continuous Adequacy Assessment. Building on SuMo, this thesis introduces ReSuMo, the first regression mutation testing approach and tool for Ethereum smart contracts [15, 24]. ReSuMo focuses mutation testing on contracts and tests potentially affected by the latest code changes, reducing the high computational cost of continuous adequacy assessment. The flexibility of regression mutation testing allows it to be employed after every commit, at periodic checkpoints, or between contract releases, the latter being particularly relevant for Upgradeable Smart Contracts. As such, ReSuMo makes mutation testing more practical throughout the development life-cycle, allowing developers to evaluate and improve test suites in parallel with the evolution of their smart contracts.

RO2: Supporting Mutation Testing in Smart Contract Auditing

While mutation testing has demonstrated its effectiveness in test adequacy assessment, its adoption in smart contract auditing workflows remains limited. Auditors play a central role in ensuring the security and reliability of smart contract-based applications before release, yet they rarely incorporate mutation analysis due to its perceived cost and lack of direct benefits. This thesis aims to explore how mutation testing can enhance smart contract audits and support its broader adoption. The contributions in this area focus on three key aspects: (1) investigating the role of live mutants in manual contract reviews,

(2) streamlining mutation testing for audit processes, and (3) leveraging automated approaches to derive test cases from surviving mutants.

Investigation of Mutant Productivity in Solidity Audits. This thesis investigates the use of surviving (i.e., *live*) mutants in smart contract auditing processes [10], examining whether their value extends beyond test improvement alone. Prior research in industrial settings [139] suggests that analyzing live mutants can sometimes prompt code refactoring and uncover overlooked faults in the code under test. To explore this perspective, we conduct a retrospective study on projects submitted for a review to *Quantstamp*. The results suggest that certain *inspection-productive* mutants can indeed enhance knowledge and, in some cases, reveal previously reported issues within Solidity smart contracts. This thesis also further refines mutation operators to increase the proportion of productive mutants, reinforcing mutation testing as a valuable technique in smart contract auditing.

End-To-End Mutation Testing Framework for Auditing. The thesis introduces SuMo as a Service (SaaS), an end-to-end framework that streamlines and speeds-up mutation testing within auditing workflows [23]. This is important to ensure that live mutants are delivered timely for further analyses, such as manual review or the application of automated test generation techniques on live mutants. Building on the modularity of SuMo, the framework employs remote parallelization of mutation testing and significantly reduces mutation testing time, up to 70% with respect to the stand-alone SuMo module. Additionally, the framework enhances usability by incorporating a code review interface, facilitating integration into standard auditing workflows.

Automated Test Generation From Live Solidity Mutants. Next, the thesis presents Alchemist, the first framework for generating test cases from Solidity mutants using LLMs [17, 18]. Unlike traditional, direct-prompting approaches, Alchemist embeds the scientific method into the code generation process, systematically refining test cases through iterative hypothesis generation, experimentation, and observation. Results confirm that Alchemist can outperform direct-prompting strategies, enhancing the likelihood of delivering effective test cases. While generating correct Solidity test code remains challenging, Alchemist allows auditors to provide actionable test improvement suggestions with minimal effort and costs. Furthermore, although the framework is specifically designed for smart contracts, its methodology is applicable to additional testing contexts and general-purpose programming languages.

RO3: Supporting Smart Contract Maintenance Practices

While smart contracts are designed to be immutable, upgradeability mechanisms (such as the proxy pattern) allow developers to introduce new features and improvements over time. However, upgrading smart contracts carries risks, including breaking changes and behavioral regressions that could disrupt contract functionality. Despite the availability of historical transaction data recorded on the Ethereum blockchain, existing testing methodologies for smart contract upgrades remain inadequate, lacking systematic frameworks to validate changes against prior contract behavior. To address this gap, this thesis

introduces a novel capture-replay testing approach that leverages historical transactions to ensure upgrade reliability and support smart contract maintenance activities.

Transaction-based Capture-Replay Testing for Proxy-based Upgradeable Smart Contracts. The thesis presents Catana, an open-source framework that automates capture-replay testing for proxy-based USCs. Catana simulates the upgrade process locally, replays historical transactions on the new contract version, and analyzes consistency across executions [12, 16]. Catana implements a dedicated workflow to extract and analyze changes to the internal proxy storage (i.e., the state variables of the USC), offering deeper insights into state changes that may introduce regressions or break compatibility. This approach enhances observability and helps developers detect disruptive modifications, validate intentional updates, and debug issues by analyzing pre- and post-upgrade logs. The experiments we conducted demonstrate that storage data analysis significantly improves replay testing capabilities, with approximately 86.5% of detected disruptive upgrades identified solely through changes in state variables. Furthermore, an evaluation of test selection policies provides guidelines for balancing efficiency and fault-detection capabilities.

5.2 Additional Contributions

During the broader course of my PhD, I also investigated *software test flakiness*, a core issue in test automation that causes tests to exhibit intermittent outcomes when re-executed. I conducted this research as part of a funded grant with the Istituto di Analisi dei Sistemi ed Informatica "A. Ruberti" (IASI) and Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" (ISTI) of the Italian National Research Council (CNR). This work provided meaningful insights and new perspectives on test flakiness, and contributed to shaping my skills as a researcher. However, since it does not specifically focus on smart contract quality assurance topics, it is beyond the core scope of this thesis. In the following, I briefly summarize the relative publications.

Scoping Review of Software Test Flakiness

The work in [13] stems from the inconsistencies in how flaky tests are defined, classified, and understood, both in academic and industrial contexts. In recent years research aiming at understanding and mitigating the problem of test flakiness has boomed, also pushed by alarms raised by big companies as Google [115], Facebook [108] or Apple [92], among others, on the extent and cost of this phenomenon. In the fast rising of articles addressing test flakiness, researchers provide different characterizations and modelings of the involved aspects and connected techniques. The lack of a unified terminology has led to fragmented research efforts, making it difficult to compare findings and build on existing knowledge.

Motivated by this gap, we conducted a scoping review [7] of both white and grey literature, drawing from academic sources and practitioner discussions, to analyze how the concept of flakiness has been approached across different studies. We identified *key patterns* in the ways flaky tests are described, noting that while most definitions agree

that flakiness manifests as non-deterministic test outcomes, there is *little consensus* on the specific conditions under which a test should be considered flaky. Some definitions strictly require that both the code under test and the test itself remain unchanged, while others extend this requirement to include environmental factors, configurations, or execution conditions.

One of the key contributions of this study was mapping out the varying definitions and classifications to propose a *clearer vocabulary* for discussing test flakiness. In particular, we identified common themes and conflicting perspectives within flaky tests classifications, considering aspects such as *order-dependency*, *non-determinism*, *implementation-dependency* and *latency*. This work supports future research aimed at developing consistent frameworks and solutions for identifying, categorizing, and addressing test flakiness.

Analysis of Flaky Test Behavior in the Field

The work in [14] offers a new perspective on software test flakiness. Traditionally, test engineers view flakiness as a nuisance, an obstacle to reliable test automation that needs to be suppressed. Rather than treating flaky tests as purely useless code to be eliminated, we investigated whether they could, in some cases, be useful indicators of hard-to-detect failures that manifest in real-world deployments. These are failures that often stem from elusive bugs, known in the literature as *Mandelbugs* [38, 47]. In particular, we hypothesized that some flaky tests, instead of being dismissed outright, could *provide valuable insights when moved* from the development environment *into the field*. In fact, the sporadic failures that occur during pre-deployment testing may align with actual issues experienced by users post-deployment, meaning that flaky tests could serve as early warning signs for problems that are otherwise elusive in controlled testing conditions.

To explore this idea, we designed and conducted a simulation study, where we examined the behavior of flaky tests when executed in a production-like environment. In particular, we compared the behavior of real-world flaky test methods (from the International Dataset of Flaky Tests [95]) when run in a standard development setting (*In-House*) against their execution in a production (*In-Vivo*) environment. To this end we relied on Groucho [31, 30], a specialized framework for In-Vivo testing of Java programs. Groucho allows test cases to be executed within a running system, providing rollback mechanisms to ensure that they do not interfere with normal user operations. We used a benchmark application to generate realistic execution scenarios, and adapted the flaky test cases for In-Vivo execution, ensuring that they could be executed in response to real system states rather than artificial test setups.

The results of this study confirmed that moving flaky tests from the development environment to a simulated field setting can affect their outcome. While some tests that were flaky in the lab remained flaky in the field, others exhibited an increase in failure frequency, revealing that the conditions in which they were executed had a profound impact on their behavior. Additionally, some of them started from a deterministic state In-House, and only become intermittent once they were moved to the field. This evolution suggests that a field testing approach can help to (re)create those conditions necessary for the flakiness to emerge, and possibly help to identify the relative cause of failure. Our

main goal, however, was to understand whether test intermittence could be seen as a useful indicator of hard-to-detect failures rather than a negative event to be eradicated. Our analysis revealed that, in most cases, the root cause of an intermittently failing test is found in the bad design of the test code itself. However, we also found evidence of a field-intrinsic bug that remained hidden during the In-House phase due to the known problem of combinatorial explosion. This suggests that, in some instances, intermittent failures should not be ignored but rather investigated as potential indicators of deeper underlying issues.

PART II

MUTATION TESTING FOR SMART CONTRACTS

CHAPTER 6

SUMO: MUTATION TESTING FOR SMART CONTRACTS

Blockchain technologies have had a rather disruptive impact on many sectors of the contemporary society. The establishment of virtual currencies is probably the most representative case. In the last years, the introduction of Smart Contracts has further increased the potential impact of blockchain. These self-enforcing programs have interesting peculiarities, such as code immutability, that require innovative testing methodologies. This Chapter presents a mutation testing approach for assessing the quality of test suites of Ethereum Smart Contracts written in Solidity. Specifically, we propose a novel suite of mutation operators capable of simulating a wide variety of traditional programming errors and Solidity-specific faults. We implemented our approach in a reference framework called SuMo, and we evaluated its effectiveness on a set of real-world Solidity projects. Results highlighted a low recurrent mutation score for the test suites shipped with the selected applications. Moreover, analyzing the surviving mutants of a selected project helped us to identify faulty test cases and Smart Contract code. These results suggest that SuMo can concretely improve the fault-detection capabilities of a test suite, and help to deliver more reliable Solidity code.

This chapter is based on material from the following publications: [19, 22]

* * *

6.1 Introduction

Since their first appearance, blockchain technologies have seen a rapid and widespread adoption, and novel application scenarios are continuously emerging. On Ethereum, smart contracts permit to encode application logic directly on the blockchain. Given their immutability and high-stakes execution, smart contracts demand rigorous testing [20]. However, the range and maturity of testing methodologies and tools available for smart contracts is limited compared to those for traditional software [207]. While static

and dynamic analysis tools (e.g., Slither [61], Echidna [70]) can detect certain classes of bugs, few tools support developers in evaluating the effectiveness of their test suites. In practice, both traditional and blockchain developers often rely on code coverage to assess test adequacy [1, 27]. However, coverage alone is a poor proxy for test quality [81, 182]. Without deeper adequacy assessment techniques, developers risk overestimating the reliability of their tests and deploying unreliable contracts.

Mutation testing is a well-established and powerful technique for assessing the quality of test suites [33]. Studies have shown a strong correlation between the ability to detect mutants and the discovery of real-world bugs [88], making this technique particularly *valuable* in *business-critical domains*. Although a few approaches and tools have been proposed (*MuSC* [192], *ContractMut* [75], *Vertigo* [76], *Universal Mutator* [3], *Regular-Mutator* [84] and *Deviant* [40]), mutation testing is still seen as an experimental method rather than a standard part of development.

As introduced in Section 4.2, the absence of a standardized approach to Solidity mutation testing has led to a *fragmented ecosystem* where tools are narrowly focused or only partially applicable. Different works introduce alternative sets of mutation operators, often narrowing the mutation process to specific categories of smart contract faults (e.g., access control vulnerabilities). From a practical standpoint, many implementations are not agnostic to the smart contract testing environment or focus solely on generating mutants without addressing the execution of tests. Furthermore, the generation of unproductive (e.g., equivalent, subsumed and uncompilable) mutants remains a problem. To establish mutation testing as a valuable technique in the blockchain domain, it is important to make the process both rigorous and practical for developers.

In this thesis, we propose a novel mutation testing approach and tool for Solidity smart contracts, and we implement it in a reference framework called *SuMo* (i.e., a mutation of SoMu, which stands for Solidity Mutator). We design and introduce a comprehensive set of 44 mutation operators for simulating smart contract faults, supporting the derivation of high-quality test suites. In particular, we introduce 11 novel (i.e., SuMo-specific) mutation operators, 7 of which target the unique features of Solidity. SuMo also provides users with the possibility to enable *optimized* mutation rules to limit the generation of likely subsumed mutants. We provide SuMo as an open-source npm module with the intent of supporting both practitioners and researchers interested in smart contract testing. Then, we perform a comprehensive empirical evaluation of SuMo on test suites associated with open source smart contracts on GitHub. This evaluation was guided by three key research questions:

- *RQ1: Do the SuMo-specific and Solidity-specific operators contribute to the identification of weaknesses in Smart Contract test suites?* By comparing the Mutation Score across different operator categories, we found that SuMo and Solidity-specific operators are responsible for a significant portion of undetected mutants. Solidity-specific operators *revealed important testing gaps*, particularly around language features such as event emission, function modifiers, and exception handling statements.

- *RQ2: Can operator optimizations speed up the mutation testing process with out compromising the adequacy assessment?* Empirical results demonstrated that optimized operators yield nearly identical Mutation Scores while reducing the generation of subsumed mutants and lowering time costs by $\approx 30\%$ on average. Neverthe-

less, the non-optimized operators did produce additional, unique live mutants that could be valuable in business-critical scenarios.

- *RQ3: Can the live mutants help to improve the fault-detection capabilities of a test suite?* By applying SuMo to a Solidity project, we discovered that analyzing live mutants not only guided the creation of new test cases but also uncovered a bug and refactoring opportunities in the smart contract code.

These results suggest that SuMo can be a valuable addition to the smart contract developer's toolkit and contribute to the release of more reliable code.

The rest of this chapter is organized as follows. Section 6.2 introduces SuMo and its mutation testing approach, while Section 6.3 expands on the research questions. Section 6.4 describes the implemented mutation operators in detail, while the design of the SuMo framework and its implementation is presented in Section 6.5. Section 6.6 presents the experiment methodology and setup. Section 6.7 discusses the results, providing insights into the practical application of Solidity mutation testing. Section 6.8 describes the threats to the validity of this study, while Section 6.9 presents work that is close to our proposal. Finally, Section 6.10 identifies opportunities for further research.

6.2 SuMo Mutation Testing Approach

Mutation testing evaluates the quality of a test suite by assessing its ability to detect and eliminate mutants. Consequently, the effectiveness of mutation testing depends on the design of a comprehensive set of *mutation operators* capable of simulating real-world bugs. While there is no such thing as a rule-book for designing mutation operators, prior work often imports traditional mutation rules from well-established tools, and modeled Solidity-specific mutations based on real-world faults (e.g., from GitHub issues, fault taxonomies, and vulnerability classifications). In addition to defining effective mutation operators, the generation of unproductive mutants (e.g., *uncompilable*, *equivalent* and *subsumed*, as defined in Section 3.5) must also be addressed. Such mutants can increase the time costs and manual effort involved in mutation testing, while also compromising the reliability of the assessment. In the next Section, we present the process we followed to derive SuMo's mutation operators. Then, in Section 6.2.2, we explain the strategies we employed to minimize the generation of unproductive mutants.

6.2.1 Fault Model and Mutation Operators

In mutation testing, a fault model refers to the theoretical framework that describes how a system or program might fail. Therefore, it identifies the types of fault that can be injected into a program to evaluate the effectiveness of its tests. To define the fault model for SuMo, we followed a two-step process:

1) Analysis of the Solidity Language and Fault Taxonomies: First, we conducted an in-depth analysis of the Solidity language (Version 0.7.1[1]). We considered

[1]Solidity documentation V 0.7.1 `https://docs.soliditylang.org/en/v0.7.1/`

the classification of programming aspects reported in the documentation (e.g., *Special Variables and Functions*, *Types*, etc ...), and identified mutation operator opportunities accordingly. This analysis informed the design of meaningful mutation operators that simulate faults likely to occur during contract execution, focusing on mistakes developers might make in their reasoning. Additionally, we revised available fault taxonomies [8, 53] for Solidity, and checked common known vulnerabilities as described in the Smart Contract Weakness Classification (SWC) registry [123].

2) Analysis of Previous Work on Mutation Testing: For the second step we revised mutation operators and empirical data from both traditional and domain-specific research. The Solidity language shares some traits with with widely used general-purpose languages like Java, JavaScript, and Python. For example, Solidity contracts are perfectly analogous to Java classes. They contain persistent data in state variables, functions that can modify these variables, support inheritance and polymorphism. Both languages enforce type declarations at compile time, and employ access modifiers for functions and variables. However, the syntax of Solidity is more similar to Javascript, including identical control flow constructs and the use of global objects. Thus, we reviewed mutation operators introduced in traditional, state-of-the-art tools like *muJava* [107] and *PIT* [140]. This allowed us to incorporate operators based on well-established tools, and to implement novel operators that were never applied to Solidity Smart Contracts. The remaining operators find some correspondence in existing work on Solidity mutation testing [3, 40, 76, 75, 192], although some of them were reconsidered or slightly modified. Specifically, SuMo only inherits operators that were empirically shown to be effective, while those generating a significant number of unproductive mutants were excluded. Several operators were also enhanced to simulate a broader range of potential faults, or updated to comply with the most recent Solidity version.

6.2.2 Handling Unproductive Mutants

During the design of SuMo, we performed a thorough analysis of Solidity and considered empirical data from prior work to exclude mutation rules that typically generate high ratios of unproductive mutants. However, it is not always wise to employ drastic selective mutation strategies. Indeed, the insights provided by some mutation operators are too valuable, and they cannot be excluded without compromising the quality of the test suite. Therefore, we implemented the following methods:

Detecting Equivalent and Redundant Mutants Equivalent mutants preserve the original program's semantics, meaning they cannot be detected by any test suite. Similarly, redundant mutants are functionally indistinguishable from other mutants, offering no additional insights to developers. Both types of mutants often evade automated detection techniques and must be weeded out manually, introducing additional mental overhead for the developers. To address these issues while minimizing the risk of losing meaningful information, we utilize the state-of-the-art TCE technique [133]. As introduced in Section 3.5, TCE leverages compiler optimizations to identify functionally equivalent code. This approach is not only relatively fast and straightforward to apply but has also demonstrated good results in real-world scenarios [133, 72].

Limiting Uncompilable Mutants Solidity is a compiled language, thus introducing mutations can sometimes result in compilation errors. Since each mutant must be compiled before being tested, every failed run introduces a time overhead without contributing to the assessment of the test suite. To limit the generation of such mutants, SuMo generates the AST of the target contracts (based on the Solidity language grammar [174]), and then visits the nodes according to preset rules. AST-based mutation testing has many advantages over regular-expression based approaches. The latter operate on plain text without any understanding of the programming language's rules. This can lead to nonsensical mutations and introduce many syntax errors. For example, Ivanova et al. [84], showed that a regexp-based approach generates ≈ 85% of uncompilable Solidity mutations. An AST-based approach is superior for mutation testing in terms of precision, semantic awareness, and the possibility to inject context-aware mutations. Specifically, each operator in SuMo can encode preconditions that determine whether a mutation should be injected or not based on the characteristic of the target node and its relationship with other nodes.

Limiting Subsumed Mutants A subsumed mutant is defined on the basis of a subsumption relationship [94]: a mutant *m1* subsumes another mutant *m2*, if whenever m1 is detected by a test, m2 is also detected. Thus, a subsumed mutant entails the execution of a (possibly complete) test run without providing new information to the tester, skewing the Mutation Score and increasing testing time. To mitigate their impact, SuMo provides users with selective mutation mechanisms by offering two sets of mutation operators:

1) The **Non-Optimized (Non-Opt)** operators generate a comprehensive set of mutants. However, this comes at a price. This set generates overall more mutants (and increases the likelihood of generating subsumed ones), resulting a more time-consuming mutation testing process.

2) The **Optimized (Opt)** operators consist of simplified rules that aim to limit the generation of likely subsumed mutants. In particular, we merged those mutations that are likely to be killed by the same test (in the scope of each mutation operator). The definition of the optimized rules was driven both by our knowledge of the Solidity language and by the analysis of existing mutation operators, such as the ones implemented in PIT[2]. The optimizations involve some specific mutation operators, which we better describe in sections 6.4. The main drawback is the potential loss of some useful, non-subsumed mutants. Indeed, the subsumption relationships among the mutants do not have universal validity, as they are strictly dependent on the implementation of the test suite.

6.3 Research Questions

In the following, we present the Research Questions (RQs) that we aim to answer in this thesis.

RQ1: Do the SuMo-specific and Solidity-specific operators contribute to the identification of weaknesses in Smart Contract test suites? Our main objective

[2] `https://pitest.org/`

is to assess whether the novel and the Solidity-specific operators implemented by SuMo contribute to the improvement of the test suite. To answer this question, we will analyze the Mutation Score achieved by the test suite with respect to the relevant mutation operators. The general assumption is that small values for the Mutation Score are a symptom of missing or poorly designed test cases. Nevertheless, the generated mutants could also mimic bugs that are not that dangerous or likely to occur, so that the effort spent is not worth it. Thus, we analyze and discuss the live mutants generated by each operator, paying particular attention to those that are rarely killed throughout the experiment.

RQ2: Can operator optimizations speed up the mutation testing process without compromising the adequacy assessment? RQ2 aims to evaluate the usefulness of the operator optimizations implemented by SuMo. To this end, we will repeat each mutation testing run with the Optimized and Non-Optimized operators, and we will check whether the two sets convey a similar perception of the test suites quality based on the difference between the achieved Mutation Scores. One of the major goals of operator optimizations is reducing the number of subsumed mutants; to assess the effectiveness of the simplified mutation rules, we will compare the number of subsumed mutants tested during both runs. We will also check the unique live mutants generated by the Non-Optimized operators to understand whether disabling optimizations can provide an information gain. Lastly, we will analyze the time costs of each run to understand if the optimizations can significantly speed up the testing process. In this way, we can assess whether both sets of mutation operators, Optimized and Non-Optimized, can bring benefits to the testers.

RQ3: Can the live mutants help to improve the fault-detection capabilities of a test suite? As introduced in RQ1, to assess the usefulness of the proposed operators we must consider the Mutation Score. However, live mutants are not necessarily a symptom of a bad test suite; they might remain undetected because they do not mimic real faults, or because they can only be killed by hard-to-design test cases. With RQ3 we aim to understand whether the mutants that survive a mutation testing process actually provide useful insights for improving the test suite. To be considered useful, the live mutants should help the developer to derive new test cases, or to spot weaknesses in the existing ones. A live mutant can also be considered useful if it helps to spot a bug in the code under test. To answer this question, we will run SuMo on a selected project and use the live mutants to improve its test suite, discussing those mutants that proved particularly useful in the process. We will then re-run the improved test suite to assess whether its Mutation Score actually increased.

6.4 SuMo Mutation Operators

Following the process described in Section 6.2, SuMo currently implements a set of **25 Solidity-Specific Operators** (see Table 6.2), and **19 General Operators** (see Table 6.1). In both tables, the SuMo-specific mutation operators are highlighted in bold. These 11 novel operators either target features that have not been previously considered or introduce improved mutation rules. For each operator, the tables specify the targeted

programming aspect, the availability of Optimized Rules (Opt.), and the corresponding Subsection (Sec.) where the operator is discussed in detail. In the following sections, we examine each Solidity programming aspect and present the mutation operators developed for it. Given the extensive nature of the Solidity documentation, we provide only a summary of the key concepts necessary to understand the characteristics of the operators. However, for each operator, the Table references the relevant Solidity documentation (Docs) for further details. Note that this Chapter presents the mutation operators from our work published between 2021 and 2022 [19, 22], and as such it references Solidity version 0.7.1. Many of the discussed language features still stand at the time of writing, but the operators were successively refined to keep up with the changes introduced in the latest Solidity versions.

MutOp	MutOp Name	Aspect	Opt.	Sec.	Docs
ACM	**Argument Change of overloaded Method call**	*Inheritance*	×	6.4.9	[162]
AOR	Assignment Operator Replacement	*Operators*	✓	6.4.3	[171]
BCRD	Break and Continue Replacement and Deletion	*Control*	×	6.4.2	[161]
BLR	Boolean Literal Replacement	*Types*	×	6.4.1	[176]
BOR	Binary Operator Replacement	*Operators*	✓	6.4.3	[171]
CBD	Catch Block Deletion	*Control*	×	6.4.3	[161]
CSC	Conditional Statement Change	*Control*	×	6.4.3	[161]
ECS	**Explicit Conversion to Smaller type**	*Types*	×	6.4.1	[175]
ER	**Enum Replacement**	*Types*	✓	6.4.1	[176]
HLR	Hexadecimal Literal Replacement	*Types*	×	6.4.1	[176]
ICM	Increments Mirror	*Operators*	×	6.4.3	[171]
ILR	Integer Literal Replacement	*Types*	×	6.4.1	[176]
LSC	Loop Statement Change	*Control*	×	6.4.2	[161]
OLFD	**Overloaded Function Deletion**	*Inheritance*	×	6.4.9	[162]
ORFD	Overridden Function Deletion	*Inheritance*	×	6.4.9	[163]
SKD	Super Keyword Deletion	*Inheritance*	×	6.4.9	[165]
SKI	Super Keyword Insertion	*Inheritance*	×	6.4.9	[165]
SLR	String Literal Replacement	*Types*	×	6.4.1	[176]
UORD	Unary Operator Replacement and Deletion	*Operators*	×	6.4.3	[171]

Table 6.1: SuMo Mutation Operators (General)

6.4.1 Types

Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified. Solidity provides several elementary **data types** [176], unique **data location** keywords [158], and utilities for **type conversion** [175].

Data Types

Solidity is a statically typed language designed for smart contract development on the Ethereum platform. Variables must have their types defined at compile-time, and each

MutOp	MutOp Name	Aspect	Opt.	Sec.	Docs
AVR	Address Value Replacement	*Types*	×	6.4.1	[176]
CCD	**Contract Constructor Deletion**	*Inheritance*	×	6.4.9	[156]
DLR	Data Location keyword Replacement	*Types*	×	6.4.1	[158]
DOD	Delete Operator Deletion	*Operators*	×	6.4.3	[171]
EED	Event Emission Deletion	*Events*	×	6.4.7	[160]
EHC	Exception Handling statement Change	*Special Func.*	×	6.4.8	[159]
ETR	Ether Transfer function Replacement	*Special Func.*	×	6.4.8	[159]
FVR	Function Visibility Replacement	*Modifiers*	×	6.4.6	[178]
GVR	**Global Variable Replacement**	*Special Func.*	✓	6.4.8	[155]
MCR	**Mathematical and Cryptographic function Replacement**	*Special Func.*	×	6.4.8	[167]
MOC	**Modifiers Order Change**	*Modifiers*	×	6.4.6	[170]
MOD	Modifier Deletion	*Modifiers*	×	6.4.6	[170]
MOI	Modifier Insertion	*Modifiers*	×	6.4.6	[170]
MOR	Modifier Replacement	*Modifiers*	×	6.4.6	[170]
OMD	**Overridden Modifier Deletion**	*Inheritance*	×	6.4.9	[169]
PKD	Payable Keyword Deletion	*Modifiers*	×	6.4.6	[170]
RSD	Return Statement Deletion	*Return*	×	6.4.4	[172]
RVS	**Return Values Swap**	*Return*	✓	6.4.4	[173]
SCEC	Switch Call Expression Casting	*Types*	×	6.4.1	[175]
SFD	Selfdestruct Function Deletion	*Special Func.*	×	6.4.8	[157]
SFI	Selfdestruct Function Insertion	*Special Func.*	×	6.4.8	[157]
SFR	**SafeMath Function Replacement**	*Libraries*	✓	6.4.10	[166]
TOR	Transaction Origin Replacement	*Special Func.*	×	6.4.8	[155]
VUR	Variable Unit Replacement	*Units*	✓	6.4.5	[177]
VVR	Variable Visibility Replacement	*Modifiers*	×	6.4.6	[178]

Table 6.2: SuMo Mutation Operators (Solidity-Specific)

type comes with its own default value. Solidity supports a range of basic types, such as booleans and integers (both signed and unsigned), along with unique types like address. More complex types, such as arrays, structs, and mappings, allow for flexible data structures, with mappings often used for key-value pairs, crucial in smart contract logic.

While Solidity provides clear guidelines on data types and their uses, mistakes in the handling of variables can lead to critical issues, such as integer overflows, erroneous Ether transfers, and mismanaged access control. For example, integer operations can cause unintended value changes due to precision errors or overflows, while improper handling of addresses can result in lost funds or unauthorized function executions. Address mismanagement, in particular, can have severe consequences, as funds may be permanently lost if sent to an incorrect or orphaned address. Similarly, errors in boolean logic can alter the control flow of critical functions, potentially bypassing security checks or failing to execute necessary operations.

The mutation operators targeting Solidity's data types are designed to simulate these faults. Table 6.3 summarizes Solidity's data types together with the relative mutation operator (MutOp) that can target the relative variables. For complex types like arrays and mappings, no dedicated operator is implemented, because potential mistakes involving

Type	Description	MutOp
`bool`	Boolean value (true/false)	BLR
`int`	Signed integer of various sizes (e.g., `int8` to `int256`)	ILR
`uint`	Unsigned integer of various sizes (e.g., `uint8` to `uint256`)	ILR
`address`	Address of a contract or wallet, with `address payable` allowing Ether transfers	AVR
`string`	String of text, stored in memory or storage	SLR
`bytes`	Binary data with fixed sizes (e.g., `bytes1` to `bytes32`)	SLR
`enum`	User-defined type with a set of named constants	ER
`array (fixed)`	Array with a fixed size, defined at compile-time	Depends on type
`array (dynamic)`	Array that can grow or shrink at runtime	Depends on type
`struct`	Custom data structure containing various types	Depends on type
`mapping`	Key-value pairs, often used to map addresses to values	Depends on type

Table 6.3: Summary of Solidity Data Types

these types can be effectively simulated by focusing on smaller, more fundamental errors, such as those involving literals. For instance, errors in accessing an array typically arise from using incorrect index values, which can be captured by testing mutations of the literal values used for indexing.

Boolean Literal Replacement (BLR): The BLR operator is designed to test boolean logic by replacing the `true` literal with `false` and vice versa. In smart contracts, boolean values often determine outcomes in functions such as access control, transaction approvals, and conditional statements. Even a small error in boolean logic can lead to significant unintended behavior, such as granting unauthorized access or failing to execute important operations.

Integer Literal Replacement (ILR): The ILR operator targets integer literals within a contract, replacing each one by incrementing or decrementing its value by one. Solidity relies heavily on numeric types to represent quantities like token balances, time durations, or voting counts. Even small off-by-one errors in these integers can lead to major issues, such as underflows or overflows, incorrect resource allocation, or logic flaws in loops and conditionals.

Hexadecimal Literal Replacement (HLR): The HLR operator focuses on hexadecimal literals, which are commonly used in smart contracts for handling low-level operations such as memory management, bitwise computations, and address manipulation. The HLR operator replaces each hexadecimal literal with either a zero or a non-zero value.

String Literal Replacement (SLR): The SLR operator tests the contract's handling of string data by replacing any string literal with an empty string. In smart contracts, strings are often used in error messages, events, or metadata fields, and while strings do not directly influence contract logic, improper handling of them can cause confusion or disrupt the interaction between the contract and external systems like decentralized applications (dApps). For example, error messages might not convey the correct information to developers, or event logs might become misleading.

Address Value Replacement (AVR): The AVR operator in SuMo is designed to prevent errors related to the use of addresses, which are essential for various operations in Solidity. Mistakes involving addresses can propagate in unexpected ways, leading to significant issues. For instance, an incorrect address might result in a call to the wrong contract or a non-existent one. If the faulty address corresponds to an existing contract, the invoked function may not match any identifier in the recipient contract, causing either a transaction failure or the execution of a fallback function (a phenomenon known as *Call To the Unknown*). In cases where the address is invalid or orphaned, the transaction will fail, which is usually harmless unless Ether is involved. Sending Ether to an orphaned address leads to a permanent loss of funds. AVR mutates all instances of addresses in a contract, including hardcoded addresses, address variables, and special address-related function calls such as `address(this)`. The mutation is applied to both address declarations and assignments. Specifically, addresses are systematically replaced with the contract's own address (`address(this)`), the zero address (`address(0)`), or, when possible, another hardcoded address present in the contract. This approach helps identify potential issues caused by incorrect address handling.

Enum Replacement (ER): The novel Enum Replacement (ER) operator is designed to ensure the correct usage of Enums, which are commonly employed in smart contract development to create user-defined types. An Enum can consist of one or more members, with the first member serving as the default value. If a developer is unaware of this default behavior, it may lead to incorrect assumptions about the value stored in a variable. Additionally, assigning an incorrect value to an Enum variable can result in unintended contract behavior. Like other operators, ER allows the tester to choose between extended and simplified mutation rules. By default, the ER operator always swaps the first and second member of an Enum definition. This mutation forces the second members to become the default value of the Enum. As shown in Listing (Listing 6.1), ER also replaces the member of an Enum assignment with each different member. If the optimizations are enabled, each member is only replaced with a single alternative.

Conversions

Conversion mutation operators do not directly target literals or special variables but instead inject faults into type casting and conversion function calls. Type conversions, particularly in statically typed languages like Solidity, are often necessary when handling arithmetic operations, interacting with external contracts, or optimizing storage usage. However, improper type conversions can introduce serious issues, such as data truncation and precision loss. One of the most critical issues arises when casting values to a smaller

```
1 enum StateType { Created, InUse }
2 StateType public State;
3
4 function setState() public {
5   ⊙ State = StateType.InUse;
6   ⋆ State = StateType.Created;
7 }
```

Listing 6.1: Enum Definition mutated by the ER Operator

type, as this can lead to truncation faults that silently alter the intended behavior of a contract.

Explicit Conversion to Smaller type (ECS): Mistakes in integer and byte type conversions can lead to dangerous truncation faults. The novel ECS operator simulates such faults by forcing conversions to the smallest possible type (see Listing 6.2, causing the loss of higher-order bits in integers. Similarly, when applied to byte conversions, it forces the conversion to a smaller byte size, leading to truncation of the right-most bits. This can result in corrupted data or unexpected behavior in smart contracts.

```
1   uint16 a = 0x1234;
2 ⊙ uint32 b = uint32(a); // b is 0x00001234
3 ⋆ uint32 b = uint8(a); // b is 0x34
```

Listing 6.2: Explicit Conversion mutated by the ECS Operator

Switch Call Expression Casting (SCEC): The SCEC operator, originally from Deviant [40] and adopted by SuMo, targets address casting in smart contracts. It swaps two address instances that are being cast to different contract types, effectively making each contract point to the wrong address. This mutation simulates scenarios where contracts interact with incorrect addresses, leading to misdirected function calls or unintended operations.

Data Location

In Solidity, each reference type is associated with a data location keyword, which specifies where the variable is stored. The keyword `memory` is used for storing temporary variables that exist only during the execution of a function, while `storage` is used for storing state variables that persist beyond function calls. Proper usage of data location keyword is critical because it impacts memory persistence, gas costs, and overall contract behavior.

Developers coming from traditional object-oriented languages, where memory management is handled implicitly, may find the explicit mechanisms of Solidity unintuitive. Declaring a variable as memory instead of storage, or vice versa, can lead to silent failures that are difficult to diagnose. These issues are particularly relevant in functions

that modify contract state, where an incorrect data location can result in unexpected overwrites or lost updates as well as unnecessary gas costs.

Data Location keyword Replacement (DLR) The DLR operator swaps the `memory` and `storage` data location keywords, altering the behavior of the affected variable. This operator is particularly useful for simulating errors that arise from incorrect assumptions about variable persistence, ensuring that contracts properly handle state modifications while optimizing gas efficiency. Replacing storage with memory limits the lifetime of the variable to the execution of the function in which it is declared, leading to lost updates and unintended contract state behavior. Conversely, swapping memory with storage extends the lifetime of the variable, which may cause unintended persistence, higher gas costs, or even unauthorized modifications of contract data.

6.4.2 Control Structures

Solidity supports a wide range of common control structures [161], including `if`, `else`, `while`, `do`, `for`, `break`, `continue` and `try/catch`. Errors in control structures can significantly affect contract behavior by causing conditional blocks to execute at the wrong time, loops to iterate incorrectly, or exceptions to be mishandled. Faults in conditional statements (`if/else`) are typically caused by improper use of conditional or relational operators, or typographical errors in literals. These mistakes lead to incorrect execution paths, either skipping critical operations or executing unnecessary ones. Loop structures like `while` and `for` are similarly vulnerable to faulty conditions, which may cause the loop to iterate the wrong number of times or even trigger *out-of-gas* exceptions at runtime due to excessive iterations. Misuse of `break/continue` keywords can also disrupt the intended control flow of loops by skipping iterations or terminating loops prematurely. To help developers avoid these faults, SuMo implements several mutation operators targeting control structures.

Conditional Statement Change (CSC): The CSC operator mutates conditional statements by replacing the entire condition with either true or false. This forces the conditional block to either always execute or never execute, simulating logic errors in conditional expressions. CSC also mutates else blocks by deleting them, following the pattern of the classic SDL (Statement Deletion [50]) operator, effectively removing entire alternative execution paths from the contract.

Break and Continue Replacement and Deletion (BCRD): The BCRD operator swaps the `break` and `continue` statement within a loop. Replacing break with continue results in skipping the current iteration instead of terminating the loop, while replacing continue with break causes the loop to terminate prematurely. Additionally, BCRD removes instances of both break and continue to simulate their absence, potentially causing loops to behave unpredictably or not function as intended.

Loop Statement Change (LSC): The LSC operator targets `for` and `while` loops by altering their conditional statements to either true or false. This mutation forces loops to either run indefinitely (causing an infinite loop) or not execute at all. Both outcomes

can drastically affect the contract's behavior and are useful for testing how the contract handles edge cases in loop execution.

Catch Block Deletion (CBD): The CBD operator removes a `catch` block from a `try/catch` statement. This mutation is applied only when there are at least two `catch` blocks, as removing the only `catch` block would result in a compilation error. When a `catch` block is deleted, the exception handling logic—such as logging errors or performing corrective actions—will not be executed, potentially leaving the contract vulnerable to silent failures or unhandled exceptions.

6.4.3 Operators

The operators supported by Solidity are similar to the ones available in JavaScript [171]. SuMo implements several mutation operators that target unary, binary and assignment operators to simulate common errors in smart contract development.

Binary Operator Replacement (BOR): The BOR operator targets all binary arithmetic, relational, conditional, bitwise, and shift operators available in Solidity. The Non-Optimized version of BOR implements the same rules used by many related works: it swaps each instance of a binary operator with all the valid operators belonging to the same class. Since binary expressions are extremely common in any language, this approach tends to generate large amounts of mutants. Therefore, we implemented Optimized mutation rules that replace each binary operator with a maximum of two different operators of the same class. These were inspired by the Math[3], Negate Conditionals[4], and Conditionals Boundary [5] operators of the popular PIT tool, as they aim to limit the total number of generated and equivalent mutants. The mutation rules for the Optimized version of BOR are shown in Table 6.4:

Assignment Operator Replacement (AOR): The AOR operator targets compound assignments. AOR comes in two versions. The Non-Optimized operator replaces each compound assignment with all the compatible compound assignments and with the simple assignment. This type of mutation causes the affected variable to store the wrong value, which can lead to unexpected contract behaviour and altered execution results. The Optimized version applies a maximum of two replacements to each compound assignment. Besides reducing the test execution cost, this approach reduces the likelihood of generating redundant mutants. The optimized rules, which are shown in Table 6.5, were designed following the same pattern of PIT's Math operator.

Delete Operator Deletion (DOD): In Solidity, the `delete` keyword is used to assign the default value to a variable. SuMo implements the DOD mutation operator which was imported by *MuSC* [100]. DOD removes any instance of the `delete` keyword from a

[3]PIT Math: `https://pitest.org/quickstart/mutators/#MATH`

[4]PIT Negate Conditionals: `https://pitest.org/quickstart/mutators/#NEGATE_CONDITIONALS`

[5]PIT Conditionals Boundary: `https://pitest.org/quickstart/mutators/#CONDITIONALS_BOUNDARY`

Class	Target	Replacement(s)	Class	Target	Replacement(s)
Arithmetic	+	−	Conditional	^	&
Arithmetic	−	+	Conditional	&&	||
Arithmetic	*	/, **	Conditional	||	&&
Arithmetic	**	*	Relational	<	<=, >=
Arithmetic	/	*	Relational	>	<=, >=
Shift	<<	>>	Relational	<=	<, >
Shift	>>	<<	Relational	>=	<, >
Conditional	|	&	Relational	! =	==
Conditional	&	|	Relational	==	! =

Table 6.4: BOR Mutation Rules

Class	Target	Replacement(s)	Class	Target	Replacement(s)
Arithmetic	+ =	− =, =	Bitwise	^=	& =, =
Arithmetic	− =	+ =, =	Bitwise	& =	| =, =
Arithmetic	/ =	* =, =	Bitwise	| =	& =, =
Arithmetic	* =	/ =, =	Bitwise	<<=	>>=, =
Arithmetic	% =	* =, =	Bitwise	>>=	<<=, =

Table 6.5: AOR Mutation Rules

contract, one at a time. This mutation prevents a variable from assuming its default value.

Increments Mirror (ICM): Lastly, the ICM operator simulates a known typographical error[6] by replacing an increment (or decrement) shortcut assignment with its mirror. If a developer accidentally types =− instead of −=, the statement is interpreted in the wrong way. In fact, =− is an assignment followed by the unary -, while −= is a shortcut assignment which performs a decrement operation. The usage of this operator for Solidity was first introduced by Vertigo [76], which imported it from PIT. However, since the unary + is no longer supported by Solidity, the ICM operator was modified to only target decrements.

Unary Operator Replacement and Deletion (UORD): The UORD mutation operator targets all unary arithmetic, bitwise and conditional operators. Each unary operator is either removed or replaced with a different operator of the same class. Unary expressions are subject to three types of mistakes: replacement, omission and insertion. However, related works such as MuSC [192] show that inserting unary operators generates elevate amounts of equivalent mutants. As shown in Table 6.6, UORD deletes the logical NOT (!), bitwise NOT (~), and unary − to simulate omission. The mutation rules that target increments

[6]SWC-129: `https://swcregistry.io/docs/SWC-129/`

Class	Target	Replacement(s)	Class	Target	Replacement(s)
Arithmetic	++	−−, *deletion*	Conditional	!	*deletion*
Arithmetic	−−	++, *deletion*	Bitwise	~	*deletion*
Arithmetic	−	*deletion*			

Table 6.6: UORD Mutation Rules

(++) and decrements (−−) are based on PIT's Increments [7] and Remove Increments [8] operators.

6.4.4 Function Parameters and Return Variables

In Solidity, functions [164] take typed parameters as input and may, unlike in many other languages, also return an arbitrary number of values as output [173]. Additionally, functions can return data using two different syntax structures (*implicit* or *explicit*). These features, while flexible, can introduce confusion and lead to mistakes in function implementation, especially when handling return values or managing how data is passed between functions.

Furthermore, in many other programming languages, a missing return statement in a function that declares a return type would trigger a compilation error. Solidity allows functions with implicitly declared return values to omit an explicit return statement without causing a compilation failure [172]. Instead, it automatically initializes return variables to their default values (e.g., 0 for integers and false for booleans). This unique behavior can lead to unintended logic errors, as developers coming from languages that enforce explicit returns (such as Python or Java) may mistakenly assume that execution will fail rather than silently returning a default value.

SuMo introduces mutation operators that target a function's return structure, but not its parameters, as mutating function parameters at the declaration level can generate a high number of uncompilable mutants. Instead, it is more effective to target the arguments passed to function calls during contract execution by means of other mutation operators (e.g., those that target types, as described in Section 6.4.1).

Return Statement Deletion (RSD): Faults in return statements may cause the contract to show unexpected behavior, especially regarding the interaction between functions and the integration with other contracts. The RSD operator is designed to ensure that return statements are thoroughly tested in Solidity functions. To this end, it removes the return statement from the body of each contract function. As shown in Listing 6.3, this causes the affected function to return the default value for its return type (or types).

Return Values Swap (RVS): SuMo also includes the novel RVS operator, which specifically targets functions that return multiple values. RVS swaps these values among themselves, encouraging in-depth testing of functions with complex return statements. The

[7] PIT Increments: `https://pitest.org/quickstart/mutators/#INCREMENTS`

[8] PIT Remove Increments `https://pitest.org/quickstart/mutators/#REMOVE_INCREMENTS`

```
1 uint public fee = 10 ;
2
3 function incrementFee(uint i) public returns(uint) {
4  fee += i;
5  ⊙ return fee; //this returns fee
6  ⋆ //return fee; //this returns 0
7 }
```

Listing 6.3: Return Statement mutated by the RSD Operator

operator works with both return structures supported by Solidity: *implicit*, where the return values are only declared as part of the function signature, or *explicit*, where a return statement is defined within the function body. In its Non-Optimized form, RVS swaps all compatible return values in a function, while the Optimized version swaps each value with only one other compatible value. Compatibility is determined by data type, preventing the generation of invalid mutants and improving testing efficiency. As shown in Listing 6.4, in a function like `signatureSplit`, which returns three values — `uint8 v, bytes32 r, bytes32 s` - RVS only swaps the two `bytes32` return values.

```
1  function signatureSplit(bytes memory signatures, uint256 pos) internal
       ↪ pure
2  ⊙ returns (uint8 v, bytes32 r, bytes32 s) {
3  ⋆ returns (uint8 v, bytes32 s, bytes32 r){
4          assembly {
5          let signaturePos := mul(0x41, pos)
6          r := mload(add(signatures, add(signaturePos, 0x20)))
7          s := mload(add(signatures, add(signaturePos, 0x40)))
8          ...
9      }
10 }
```

Listing 6.4: Return Statement mutated by the RVS Operator

6.4.5 Variable Units

Solidity provides two types of variable units: Ether Units and Time Units [177]. These units are used for smart contract calculations related to currency and time. A numeric literal can take a suffix of `wei`, `gwei` or `ether` to denote different sub-denominations of Ether, while time-related calculations can use units from `seconds` to `weeks` to specify durations in a human-readable manner. Faults in Ether unit specification can lead to drastic miscalculations in value transfers, and are particularly dangerous in functions that involve payments, withdrawals, or token exchanges. Confusing ether units results in an exchange discrepancy of 10^9, potentially leading to either a massive overpayment or an unintended loss of funds. Similarly, using an incorrect time unit can lead to anomalous contract behavior. If a time unit is mistakenly set to `days` instead of `seconds`, an action

that should execute within moments could be delayed. This can be especially problematic in time-sensitive smart contracts, such as those handling auctions or voting processes. If the affected value acts as a trigger for executing key operations, a misconfigured time unit could result in logic executing too soon, too late, or not at all.

Variable Unit Replacement (VUR): The VUR mutation operator simulates the incorrect usage of variable units by replacimg each ether and time unit suffix with a different, compatible suffix. For example, Listing 6.5, demonstrates an Ether unit mutation. The original Solidity code checks whether the amount is below 70 wei before executing a transaction. The VUR operator mutates this condition by replacing wei with gwei, drastically altering the threshold. While 70 wei represents a minuscule fraction of Ether, 70 gwei is significantly larger, potentially preventing the condition from ever being met. As a result, the happy-path may fail to execute as expected, blocking legitimate withdrawals or failing to trigger other critical operations.

```
1 function withdrawAmount() external {
2 ⊙ if(amount < 70 wei) {
3 ⋆ if(amount < 70 gwei) {
4     // ...  withdraw logic ...
5   }
6 }
```

Listing 6.5: Ether Unit mutated by the VUR Operator

6.4.6 Function Modifiers

In Solidity, modifiers allow developers to alter the behavior of functions in predefined ways, such as restricting access, controlling data visibility, or specifying conditions that must be met before the function executes. A function definition can be extended with built-in modifiers, like those for **visibility** [178] and **state mutability** [170], which define the accessibility and operational constraints of a function. Furthermore, developers can define **custom modifiers** [170] and attach them to a function (as a `modifier-invocation`) to extend its functionality, often for tasks like access control or precondition checks.

State Mutability Modifiers

Solidity provides three state mutability modifiers (`pure`, `view` and `payable`) which define how functions interact with the blockchain's state (see Table 6.7). The pure and view modifiers restrict functions from modifying or persisting data on the blockchain, while payable determines whether a function can receive Ether from transactions. Mutations targeting pure and view have been explored in previous tools, such as Deviant [40] and MuSc [192]. However, such mutations frequently result in stillborn mutants, where the modified function's behavior does not align with the specified mutability, and they rarely affect the core logic of the contract. Thus, SuMo exclusively implements mutations

Keyword	Description
`view`	The function can read but not modify the blockchain's state.
`pure`	The function can neither read nor modify the blockchain's state.
`payable`	The function is allowed to receive Ether.

Table 6.7: Function State Mutability Modifiers in Solidity

for the payable modifier, which has a direct impact on a contract's ability to process financial transactions.

Payable Keyword Deletion (PKD): The PKD operator removes the `payable` state modifier from a function signature. Without this modifier, the function cannot receive Ether even if called with a value in a transaction. This mutation is applied both to standard functions and to the fallback, a special function that is invoked when the contract receives a plain Ether transfer or when an invocation does not match any existing function. Consider the example in Listing 6.6, where PKD is applied to a function that was originally marked as payable. Here, the deposit function, originally capable of receiving Ether, is now unable to accept funds. Any transaction attempting to send Ether to this function will fail, preventing deposits from being processed. This could disrupt financial flows within the contract, making it impossible for users to add funds, stake tokens, or participate in any financial interactions. Note that PKD does not mutate virtual, overridden, and receive ether functions to avoid compilation errors. Furthermore, unlike other tools (e.g., Deviant [40]), SuMo does not implement an insertion fault (i.e., adding the payable keyword to functions), as it is impractical to test, requiring the simulation of Ether transfers to all non-payable contract functions.

```
1 ⊙ function deposit() external payable {
2 ★ function deposit() external ~~payable~~ {
3     balances[msg.sender] += msg.value;
4 }
```

Listing 6.6: Function mutated by the PKD Operator

Visibility Modifiers

Solidity includes visibility modifiers that determine how functions and state variables can be accessed by other contracts and users. As shown in Table 6.8, visibility keywords define the access level of functions and variables. State variables are persistently stored on the blockchain, and functions interact with these variables to read or modify the contract's state. Proper use of these keywords is critical to ensure the correct operation of a contract, as inappropriate visibility settings can result in security vulnerabilities or integration issues. For instance, using overly restrictive visibility might prevent necessary interactions between contracts, while making a function too permissive can allow unauthorized access.

Keyword	Valid For	Description
`private`	Function, State Var.	Can only be accessed by the defining contract.
`internal`	Function, State Var.	Can only be accessed by the defining contract, and any contract derived from it.
`external`	Function	Can only be accessed by external contracts or users.
`public`	Function, State Var.	Can be accessed by the defining contracts, by any other contract and users.

Table 6.8: Visibility Modifiers in Solidity

SuMo implements two mutation operators that simulate faults concerning both function and variable visibility.

Function Visibility Replacement (FVR): The FVR operator replaces a function visibility keyword with a different one. There are some exceptions to this rule that we introduced to limit the possible generation of stillborn mutants. In particular, SuMo's implementation for FVR does not mutate the visibility for receive Ether functions and the fallback functions, as they can only be declared as external, and a mutation will lead to a stillborn mutant. For the same reason, it does not change the visibility of payable functions to internal or private. Lastly, FVR does not delete the visibility keyword, as the function default visibility is no longer allowed.

Variable Visibility Replacement (VVR): The VVR operator replaces the visibility keyword of a state variable with a different one. VVR also adds the private and public keywords to those variables that have the default visibility (which is equal to internal).

Custom Modifiers

In addition to built-in modifiers, Solidity allows developers to create custom modifiers that extend the behavior of functions. Similar to aspects in aspect-oriented programming, these custom modifiers are defined separately from the function logic and can be used to enforce preconditions or add specific functionality without duplicating code. Errors in the implementation of custom modifiers, or in the usage of modifiers imported from existing libraries, can introduce serious faults. Some widely used libraries that provide these modifiers include *OpenZeppelin* and other protocol-specific contracts, which are frequently used to enforce essential constraints:

- **Access Control Modifiers** (e.g., `onlyOwner`, `onlyAdmin`, `onlyRole`): These restrict function execution to specific addresses, typically contract owners or administrators. If incorrectly applied, they may block authorized users from executing necessary transactions or fail to prevent unauthorized access, enabling privilege escalation attacks.
- **State-Dependent Modifiers** (e.g., `whenNotPaused`): These ensure that functions only execute under specific contract states, such as pausable contracts or

time-based logic in auctions and token sales. A misapplied state-dependent modifier may freeze functionality unintentionally or allow actions at the wrong time.

- **Reentrancy Protection** (`nonReentrant`): Used to prevent reentrancy attacks, where malicious contracts repeatedly call back into a function before previous executions complete (e.g., the infamous DAO hack). Omitting this modifier in vulnerable functions could expose the contract to theft of funds.

- **Payment Restrictions** (e.g., `requiresMinimumDeposit`): Often used to enforce a minimum amount of Ether or tokens in a transaction. Removing this kind of condition could lead to underfunded deposits, disrupting contract logic.

Since modifiers dictate when and how a function executes, errors in their implementation can introduce serious security vulnerabilities or logic faults. Attaching the wrong modifier to a function may result in unnecessary restrictions, blocking legitimate function executions (SWC-123). Conversely, omitting a necessary modifier might allow unauthorized or malicious operations, such as fund withdrawals or contract destruction. SuMo includes five operators for custom modifiers: MOD, MOI and MOR were inspired by *Deviant* [40] and *ContractMut*[75], whereas MOC and OMD (described in Section 6.4.9) introduce novel mutation rules.

Modifier Deletion (MOD): The MOD operator (Listing 6.7) removes the modifier attached to each function signature, one at a time. This prevents the logic implemented by the modifier from running upon the execution of the function.

```
⊙ function updatePrice(uint newPrice) onlyOwner {
★ function updatePrice(uint newPrice) ~~onlyOwner~~ {
  price = newPrice;
}
```

Listing 6.7: Function Modifier mutated by the MOD Operator

Modifier Insertion (MOI): The MOI operator applies an existing modifier to a function that does not have any. To avoid generating too many irrelevant mutants, MOI evaluates all modifiers in the contract to check if they can be applied to the target function. Compatibility is ensured by matching modifier parameters with the function's parameters, which helps prevent the creation of stillborn mutants.

Modifier Replacement (MOR): The MOR operator replaces the modifier attached to a function with a different modifier. This mutation can alter the restrictions applied to the function, and cause the execution of unexpected operations.

Modifiers Order Change (MOC): Lastly, MOC is a novel operator that targets functions with multiple modifiers. It changes the order in which the modifiers are applied, which

can significantly impact the execution of the function, as Solidity enforces modifiers sequentially. A change in modifier order can, for example, make the function unreachable or lead to unintended behavior.

6.4.7 Events

Solidity contracts can leverage the Ethereum Virtual Machine (EVM) logging capabilities by firing events [160]. Whenever an event is emitted its arguments are memorized in the transaction log, a special data structure that resides in the blockchain. Even though the log is not directly accessible to contracts, it is often used by *DApps* (Decentralized Applications) to monitor the smart contract behavior and trigger code execution. Although events do not directly impact contract logic, they should still be tested carefully, as faults in emission statements can lead to issues with off-chain systems, potentially disrupting DApp functionality or obscuring important contract activities.

Event Emission Deletion (EED): The EED operator comments out an event emission statement to prevent its execution (See Listing 6.8). This mutation encourages the definition of test cases that include conditions on events. SuMo does not include a mutation operator that swaps different event emissions within the contract. Indeed, this type of mutation can generate many redundant mutants. Moreover, since events often log the arguments of the function from which they are emitted, swapping event emissions has a high chance of generating a stillborn mutant.

```
1 function payout(address winner, uint256 amount) internal{
2   // payout logic ...
3   ⊙ emit Payout(winner, amount);
4   ★ //emit Payout(winner, amount);
5 }
```

Listing 6.8: Event mutated by the EED Operator

6.4.8 Special Variables and Functions

Solidity offers many globally available variables and functions. These include properties related to blocks and transactions [155], mathematical and cryptographic operations [167], contract related functions [157], error handling functions [159] and methods for the transfer of Ether [168], the native cryptocurrency of Ethereum.

Error Handling

Solidity provides three distinct state-reverting functions to manage exceptions. Since transactions in Ethereum are atomic, any error must trigger a full rollback of all state changes to prevent unsafe contract executions. When an error occurs, these functions revert all modifications within the current call and any nested sub-calls, ensuring that the contract remains in a consistent state and signaling the failure to the caller. Specifically:

- `require(condition, message)` is used for validating conditions that must be met at runtime (e.g., user permissions, balance checks, contract states). If the condition fails, the transaction reverts and the unused gas is refunded to the sender.
- `assert(condition)` is strictly used for internal error checking and invariants (e.g., ensuring a mathematical property always holds). Upon failure, the transaction fails and the state changes are reverted without refunding gas to the sender.
- `revert(message)` is similar to require(), but often used to handle business logic errors explicitly. It provides a flexible way to abort execution, particularly in complex conditional flows.

Misuse of exception-handling functions can introduce serious vulnerabilities and unexpected behaviors in smart contracts. First, `require` and `revert` are meant for runtime validation, and should not be used for enforcing invariants or detecting internal errors. Using them incorrectly may allow faulty logic to pass unnoticed. Similarly, since `asssert` does not refund gas, using it to check user input wastes gas unnecessarily. Most importantly, missing or buggy error handling statements may cause the contract not to enforce critical rules, leading to unexpected operations or unverified assumptions.

Exception Handling statement Change (EHC): The EHC operator targets exception-handling statements to evaluate how well a test suite evaluates the smart contract failure scenarios. Specifically, EHC applies two types of mutations. First, it deletes any `require`, `assert` and `revert` statement, one at a time, preventing the condition check from executing. This forces testers to validate unhappy paths more thoroughly. Consider the Solidity function in Listing 6.9, which restricts purchases based on the amount of Ether sent. Here, EHC comments out the `revert` statement, meaning execution continues even when insufficient funds are provided. To kill this mutant, the tester is forced to evaluate the transaction reverting scenario, ensuring that the contract never allows underpriced purchases. EHC also swaps the run-time `require()` statement with `assert()` and vice versa, introducing misaligned exception behavior. Swapping statements determines whether a smart contract consumes all gas when encountering a failure, and can possibly lead to unexpected contract failures.

```
1 function buy(uint amount) public payable {
2   if (amount > msg.value / 2 ether)
3     ⊙ revert(``Not enough funds provided.");
4     ⋆ //revert(``Not enough funds provided.");
5 }
```

Listing 6.9: Revert Statement mutated by the EHC Operator

Block and Transaction Properties

Solidity provides many special variables and functions which always exist in the global namespace and are mainly used to obtain information about transactions and

Target Description	Target	Replacement(s)
Current block timestamp	`block.timestamp`	`block.difficulty`, `block.number`
Current block number	`block.number`	`block.timestamp`, `block.difficulty`
Current block mining difficulty	`block.difficulty`	`block.timestamp`, `block.number`
Address of the block miner	`block.coinbase`	`tx.origin`, `msg.sender`
Hash of the given block number	`blockhash()`	`msg.sig`
Maximum gas allowed in the block	`block.gaslimit`	`tx.gasprice`, `gasleft()`
Amount of remaining gas in the transaction	`gasleft()`	`block.gaslimit`, `tx.gasprice`
Gas price specified by the transaction	`tx.gasprice`	`block.gaslimit`, `gasleft()`
Amount of Ether sent with the message	`msg.value`	`tx.gasprice`

Table 6.9: GVR Mutation Rules

the blockchain. For example, the `block.timestamp` retrieves the current block's timestamp, while the `tx.origin` represents the address that initiated a transaction call. Improper usage of global variables can lead to severe issues. For instance, `block.timestamp` can be influenced by miners and should not be used as a source of randomness or to trigger time-dependent events. SuMo implements two mutation operators targeting block and transaction properties.

Global Variable Replacement (GVR): The novel GVR mimics the incorrect usage of the global variables and functions available in Solidity. For example, it permits to simulate the manipulation of a global variable by a malicious miner, which can prevent the developer from inserting dangerous dependencies into the contract (such as using `block.timestamp` as a source of randomness). The GVR operator swaps each global variable (or function) in the contract with one or more different variables. In particular, the Non-Optimized version of GVR swaps all the compatible global variables with each other to maximize the variety of faults injected into the contract. Since this approach can generate a considerable amount of mutants, we also designed an Optimized version of GVR. The latter only swaps those variables that are more likely to be confused during development, and only if they return a compatible data type. The relative mutation rules are listed in Table 6.9.

Transaction Origin Replacement (TOR): The TOR operator is similar to GVR, but it targets a subset of global variables for retrieving the sender of a transaction. In solidity, `msg.sender` identifies the sender of the current transaction call, `tx.origin` retrieves the root of the call chain. Confusing the two variables can cause a developer to obtain the wrong address. If the fault exists within an authorization check, it can result in unreachable code as well as unauthorized access. TOR swaps any instance of the `msg.sender`

variable with `tx.origin`, and vice versa. If the transaction is part of a chain, this mutation causes the contract to obtain a different address than intended.

Mathematical and Cryptographic Functions

Solidity offers a range of built-in function for mathematical and cryptographic operations, which are essential for secure and efficient smart contract execution. Among these are the cryptographic hash functions such as `keccak256()`, `sha256()` and `ripemd160()`. Instead, the mathematical functions `addmod()` and `mulmod()` perform modulo addition and multiplication. Because these functions take identical parameters, a developer might mistakenly interchange them, resulting in incorrect calculations.

Mathematical and Cryptographic function Replacement (MCR): The novel MCR operator implemented by SuMo replaces a mathematical or cryptographic function with a different function of the same type. For example, `addmod()` may be replaced with`mulmod()`, and `keccak256()` might be replaced with `sha256()`. This type of mutation can help to reinforce the original test suite and ensure that mathematical and cryptographic operations are performed correctly.

Contract Related Functions

Solidity provides a special *selfdestruct* function that allows to remove a smart contract from the blockchain. When executed, the `selfdestruct(address recipient)` method sends all the Ether stored in the contract to the recipient address. The contract's storage data is then permanently erased from the blockchain. Improper use of selfdestruct can lead to severe consequences. If the function is unintentionally triggered or lacks proper access control, an attacker or even an internal function call could permanently delete the contract and transfer all its funds. If selfdestruct is unreachable due to logic errors, or if it is entirely omitted, developers may lose the ability to remove a faulty or vulnerable contract from the blockchain. This can be particularly problematic in cases where a contract contains critical security flaws or requires upgrades that necessitate its removal. To assess and mitigate these risks, SuMo introduces two mutation operators that simulate selfdestruct-related faults.

```
1 function destroy() public onlyOwner {
2   ⊙ selfdestruct(owner);
3   ⋆ //selfdestruct(owner);
4 }
```

Listing 6.10: Selfdestruct Function mutated by the SFD Operator

Selfdestruct Function Deletion (SFD): The SFD removes an invocation of the selfdestruct function by commenting it out, as illustrated in Listing 6.10. This mutation prevents the contract from being removed from the blockchain, ensuring that test cases explicitly verify

whether the selfdestruct call is reachable and executed as intended. This helps to ensure that self-destruction is correctly integrated into the contract logic.

Selfdestruct Function Insertion (SFI): The SFI operator simulates a premature selfdestruct by relocating this statement to the beginning of its enclosing function. This simulates scenarios where access control measures are bypassed, or where accidental selfdestruct calls occur before other state-changing operations. SFI mutants help to thoroughly test those complex functions that must run checks or perform additional operations before the contract is actually destroyed.

Members Of Address Types

Solidity provides several methods for transferring Ether between contracts: `transfer()`, `send()`, `call()`, `staticcall()` and `delegatecall()`. The first difference among these methods is the gas limit. When transferring Ether, it is possible to allocate a certain amount of gas for code execution. For `send()` and `transfer()` the stipend is limited to 2300 gas, just enough to perform simple operations. Instead, `call()` doesn't have a bounded gas limit, which means that a larger amount of gas can be drained to complete the execution of the recipient fallback.

The second difference is the return value in case of error. The methods that do not automatically revert the transaction upon failure require a manual check for the return value. If this is not handled properly by the developer, the exception will not be propagated any further.

Lastly, Solidity provides two special variants of the message call. The `staticcall()` method is identical to a normal call, however, it does not allow any modification to the contract state. An erroneous usage of this method can prevent calls to library functions from working. The `delegatecall()` method dynamically loads code from a different address in the context of the current contract. This means that any modification to the state affects the storage of the caller, instead of the callee. A *delegate call to untrusted callee* can be extremely dangerous, as it can cause unintended state changes.

Target Description	**Target**	**Replacement(s)**
Sends Ether with limited gas, returns false on failure	`send()`	`transfer()`, `call()`
Sends Ether with limited gas, reverts on failure	`transfer()`	`send()`, `call()`
Low-level function to call another contract, allows unbounded gas	`call()`	`delegatecall()`, `staticcall()`
Performs a call that doesn't allow state modification	`staticcall()`	`call()`, `delegatecall()`
Executes code from another contract, but in the context of the caller	`delegatecall()`	`call()`, `staticcall()`

Table 6.10: ETR Mutation Rules

Ether Transfer function Replacement (ETR): The ETR operator included in SuMo helps to avoid mistakes in the usage of ether transfer functions. ETR replaces each ether transfer function with a different one. ETR is similar to operators included in other mutation testing proposals. However, in SuMo it was adapted to target the new syntax for the `call()` method, and it was augmented to mutate the `delegatecall()` and `staticcall()` methods. The replacement rules applied are shown in Table 6.10.

6.4.9 Inheritance

Inheritance [165] in Solidity allows contracts to reuse code from base contracts and enables developers to structure smart contracts more efficiently. Solidity supports key features such as constructors [156], function and modifier overriding [163, 169], and overloading [162]. These features, while powerful, introduce potential errors in contract logic due to their complexity.

Constructor

A constructor is an optional function that is run upon contract creation. The constructor allows to perform setup operations and initialize the state variables of a contract before the code is actually deployed on the blockchain.

Contract Constructor Deletion (CCD): SuMo implements a novel CCD mutation operator that targets the contract constructor. This operator was inspired by the JDC operator implemented by muJava [105], which forces a Java class to execute the default constructor. The approach described by Andesta et al. [3] is the only one that proposes dedicated mutation operators for altering a contract constructor. However, from Solidity version 0.4.22 the constructor is defined with the new `constructor` keyword, which requires the definition of a novel mutation. The CCD operator removes the constructor from a contract by commenting it out. As a result, the user-defined constructor will not be executed before deployment, and the contract state will not be correctly initialized. This mutation allows to determine whether the tests ensure the correctness of the contract state after deployment.

Overriding

Solidity's overriding mechanism allows derived contracts to redefine the behavior of functions, modifiers, and constructors from base contracts. Public and internal functions marked as `virtual` can be overridden, while `override` is used to specify that a function in the derived contract replaces the base version. When an overridden function is called, the most derived version is executed by default, but the `super` keyword or contract name can be used to invoke functions higher in the inheritance chain. Overriding is a powerful tool but can be dangerous when used incorrectly. SuMo implements several operators that were inspired by Deviant [40], the only other framework that targets Solidity's overriding mechanism. However, besides targeting overridden functions, SuMo also implements a novel operator for the modifier overriding mechanism.

Overridden Function Deletion (ORFD): The ORFD operator deletes overridden methods from derived contracts, forcing any call to the removed method to execute its base version instead. This mutation can reveal whether the tests properly distinguish between base and derived function behavior. While removing an overridden method inherited from multiple base contracts can cause compilation errors, this is a minor issue, as multiple inheritance is relatively rare in Solidity.

Super Keyword Deletion (SKD): The SKD operator deletes the `super` keyword from each function call in a contract. Without the `super` keyword, the contract will execute the most derived version of the function, bypassing the parent contract's implementation. This mutation ensures that calls to base functions are properly tested.

Super Keyword Insertion (SKI): The SKI operator inserts the `super` keyword into overridden function invocations. This mutation causes the call to execute a function of a higher level in the inheritance hierarchy.

Overridden Modifier Deletion (OMD): Lastly, OMD is a novel operator that focuses on the modifier overriding mechanism. It removes an overridden modifier from a derived contract, forcing the function to use the base modifier instead. Since base modifiers may have different restrictions or logic than their overridden versions, this mutation can cause the contract to behave incorrectly, potentially introducing security or functional flaws.

Overloading

Overloading allows a contract to define multiple functions with the same name but different numbers or types of parameters. While convenient, overloading can lead to confusion, especially if the contract is heavily overloaded. A developer might call the wrong version of a function by mistake, or redundant methods might be implemented, consuming unnecessary gas and complicating contract logic.SuMo includes two novel operators that target different aspects of method overloading in Solidity. These were inspired by the popular muJava [105, 106] mutation testing tool for the Java language.

Overloaded Function Deletion (OLFD): The OLFD operator deletes overloaded method declarations, one at a time. This mutation helps identify whether the test suite correctly distinguishes between overloaded functions. If a deleted method is not detected by the tests, it suggests that the wrong version of the function may be invoked, or that the method is unnecessary.

Argument Change of overloaded Method call (ACM): The ACM operator swaps an overloaded method call with another overloaded method that has a different number or order of arguments. This mutation simulates errors in method selection, ensuring that the usage of overloaded methods in the contract are properly tested.

6.4.10 Libraries

Libraries [166] are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused using the DELEGATECALL feature

Target	Replacement	Target	Replacement
$SafeMath.add()$	$SafeMath.sub()$	$SafeMath.mul()$	$SafeMath.div()$
$SafeMath.sub()$	$SafeMath.add()$	$SafeMath.div()$	$SafeMath.mul()$
$SafeMath.mod()$	$SafeMath.mul()$		

Table 6.11: SFR Mutation Rules

of the EVM. This means that if library functions are called, their code is executed in the context of the calling contract. Currently, SuMo implements a mutation operator that targets usages of OpenZeppelin's *SafeMath*, a popular mathematical library. SafeMath provides arithmetic functions that automatically revert a transaction in case of overflow. The reason for mutating SafeMath function calls is that developers often use them in place of the base arithmetic operators to increase code reliability.

SafeMath Function Replacement (SFR): The SFR operator swaps each SafeMath function call with all different function call. Such mutation rules permit to simulate the calculation of the wrong value. When the optimizations are enabled, SFR replaces each SafeMath function call with a single different one as shown in Table 6.11.

6.5 SuMo Design and Implementation

SuMo (a mutation of SOlidity MUtator) is a mutation testing tool for Solidity Smart Contracts [121]. The tool can automatically run a complete mutation testing process on a Solidity project, from mutant generation to test execution. The mutants are generated by 44 **mutation operators** that can be enabled and disabled by the user. In section 6.5.1 we describe the application domain of SuMo. Section 6.5.2 provides an overview of the mutation testing process, while Section 6.5.3 focuses on the design of the tool, providing insights into its main components and implementation.

6.5.1 Application Domain

SuMo is a Node.js application that operates within the typical Ethereum development ecosystem, designed specifically for mutation testing of Solidity smart contracts. We released SuMo as an npm module [120], as the majority of Solidity applications are developed within Node.js and rely on npm for dependency management. To contextualize SuMo's operational environment, in the following we describe the main components of the typical Solidity SUT.

1 - Smart Contract Files Smart contract files are the main component of an Ethereum application, containing the business logic and rules that define its behavior on the blockchain. SuMo supports mutant generation for Solidity source files.

2 - Test Files The SUT contains one or more test files that define test cases and assertions to verify contract functionality. Ethereum's testing landscape is heterogeneous

Framework	Test Language	Chain Simulator
 Brownie	Python (pytest)	Ganache
 Forge	Solidity	Anvil
 HardHat	Javascript (mocha), TypeScript	HardHat Network, Ganache
 Truffle	Javascript (mocha), Solidity	Ganache
 Hybrid	Any of the above	Any of the above

Table 6.12: Testing Frameworks supported by SuMo

and test files may be written in various languages depending on the used framework, such as JavaScript for Hardhat or Python for Brownie. These are summarized in Table 6.12, though more detailed information can be found in the official documentation of each framework.

3 - Testing Environment The testing environment for a Solidity application generally comprises two components:

- *Testing Framework*: This tool is responsible for writing and running test cases and may include built-in support for code coverage and debugging. Each framework (e.g., Hardhat, Truffle, Forge or Brownie) supports specific languages and has unique characteristics. For example, Hardhat is known for its extensive plugins and customizability, while Forge (as part of the Foundry tool-chain) focuses on performance and test execution efficiency.
- *Chain Simulator*: This component allows developers to spin up a local Ethereum-like blockchain where contracts can be deployed and tested. Chain simulators may be integrated within a testing framework (e.g., Hardhat Network in Hardhat) or managed separately (e.g., Ganache for Truffle).

To run testing activities, SuMo interacts directly with the SUT's testing environment by spawning dedicated Node processes. This approach guarantees flexibility and compatibility with most testing frameworks for Solidity (as shown in Table 6.12). Additionally, SuMo can run custom test scripts defined by the developers to evaluate hybrid test suites (e.g.: composed of Hardhat and Forge test files) in a single run.

4 - Test Configuration Another essential component of the SUT is the test configuration file (e.g., `hardhat-config.js` or `truffle-config.js`). This file allows developers to specify environment settings, network configurations, gas limits, and other parameters necessary for test execution. By directly interacting with the SUT's testing

environment, SuMo implicitly references the existing configuration, minimizing additional setup and ensuring that mutation testing is conducted under the same conditions set by the developers.

6.5.2 SuMo Workflow Overview

Figure 6.1 presents a high-level overview of SuMo's workflow, which can be divided into two main phases: Mutation Generation and Mutation Testing.

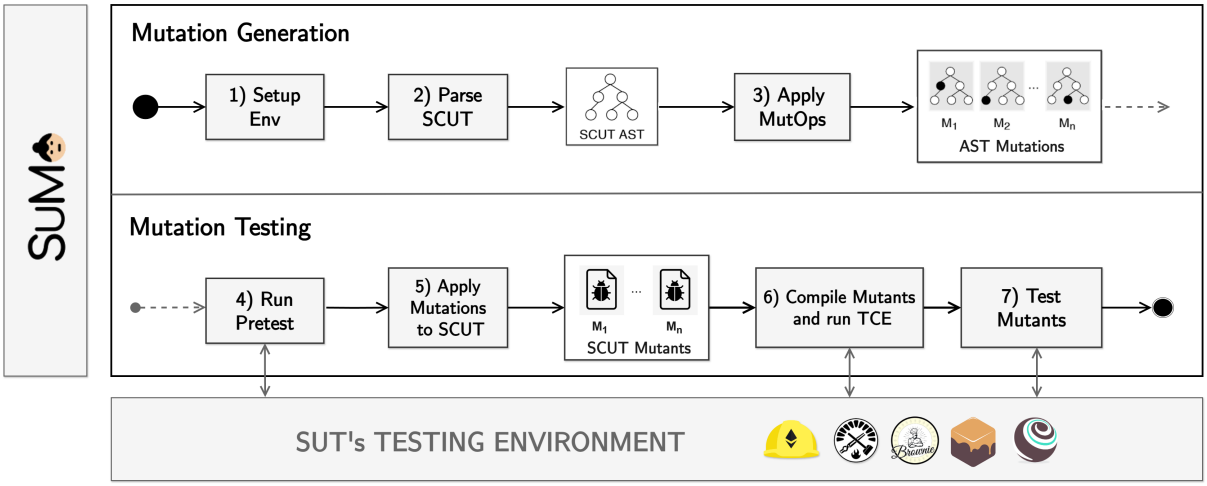


Figure 6.1: SuMo High-Level Mutation Testing Workflow

Mutation Generation The Mutation Generation phase involves creating mutation objects (in JSON format) that can be applied to the Smart Contract Under Test (SCUT). In this phase, SuMo leverages a Solidity parser[9] module to generate the AST for each SCUT. The AST is traversed, and **mutations** are generated based on rules defined by the mutation operators. As shown in the repository's documentation, users have the option to generate mutations independently from the testing phase (via the sumo lookup command) to examine the output and adjust operators as needed before proceeding with testing.

Mutation Testing In the Mutation Testing phase SuMo applies the mutations to the SCUT to evaluate the adeuqacy of the test suite. The core activities of this phase, along with their related components, are explained in detail in Section 6.5.3, but the general process involves the following steps: first, SuMo executes a pretest phase to verify that the test suite passes on the original SCUT. Then, each mutation is applied to its respective SCUT. If the mutated contract compiles, the TCE is applied to its bytecode. If the mutant is neither equivalent nor redundant, SuMo runs the test suite on it. Results, including specific details about each mutant and the overall Mutation Score, are recorded in a final **test report**. Note that any activities requiring test execution or contract compilation (such as pretest, compile, and test) are conducted via the project's testing

[9]Source code available at `https://github.com/ConsenSys/solidity-parser-antlr`

environment, which includes a testing framework (e.g., Forge) and a blockchain simulator (e.g., HardHat Network).

6.5.3 Architecture of SuMo

Figure 6.2 shows the organization of the main logical components of SuMo, In the following paragraphs we describe the characteristics of each component, and we explain how a mutation testing campaign is executed.

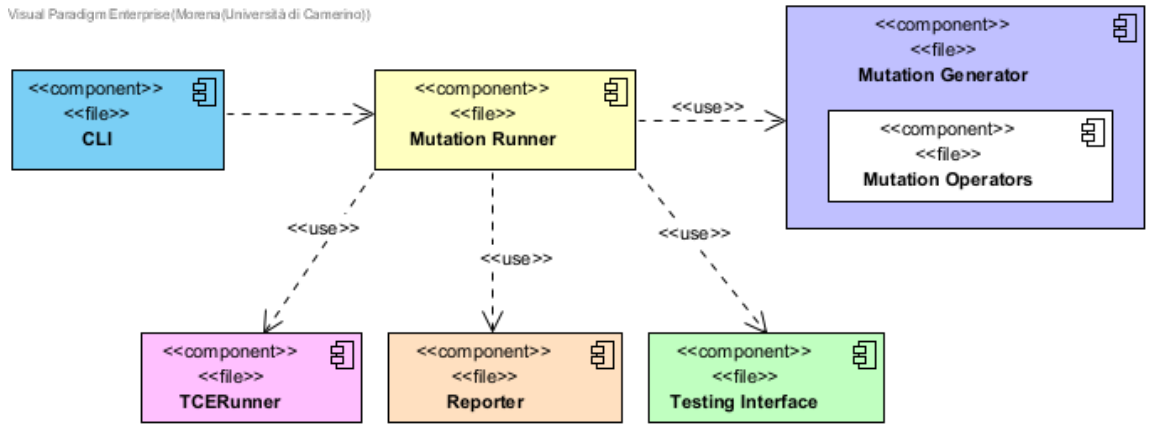


Figure 6.2: Components of SuMo

CLI

The user can interact with SuMo through a simple CLI (Command Line Interface). SuMo exposes several commands for managing the operators, for generating mutations and for starting the mutation testing process. The commands are summarized in Table 6.13, but more details can be found in the SuMo GitHub repository [121].

Cmd	Description	Usage Example
`list`	Shows the enabled mutation operators.	`$ npx sumo list`
`enable`	Enables one or more mutation operators.	`$ npx sumo enable AOR`
`disable`	Disables one or more mutation operators.	`$ npx sumo disable FVR`
`lookup`	Generates and saves the JSON mutations.	`$ npx sumo lookup`
`mutate`	Generates the mutations and saves a copy of each `.sol` mutant to ./sumo/mutants.	`$ npx sumo mutate`
`pretest`	Runs the test suite on the original smart contracts to check if all tests pass.	`$ npx sumo pretest`
`test`	Starts the mutation testing process on all mutants or on a interval of mutant hashes.	`$ npx sumo test` `$ npx sumo test mbc5e8f56 mbg5t86o6`

Table 6.13: Summary of SuMo Commands

Mutation Runner

The mutation runner is the main component of SuMo. The sequence diagram in Figure 6.3 shows how the mutation runner interacts with the other components to run a mutation testing process. When the user launches the test command, the mutation runner executes the following steps:

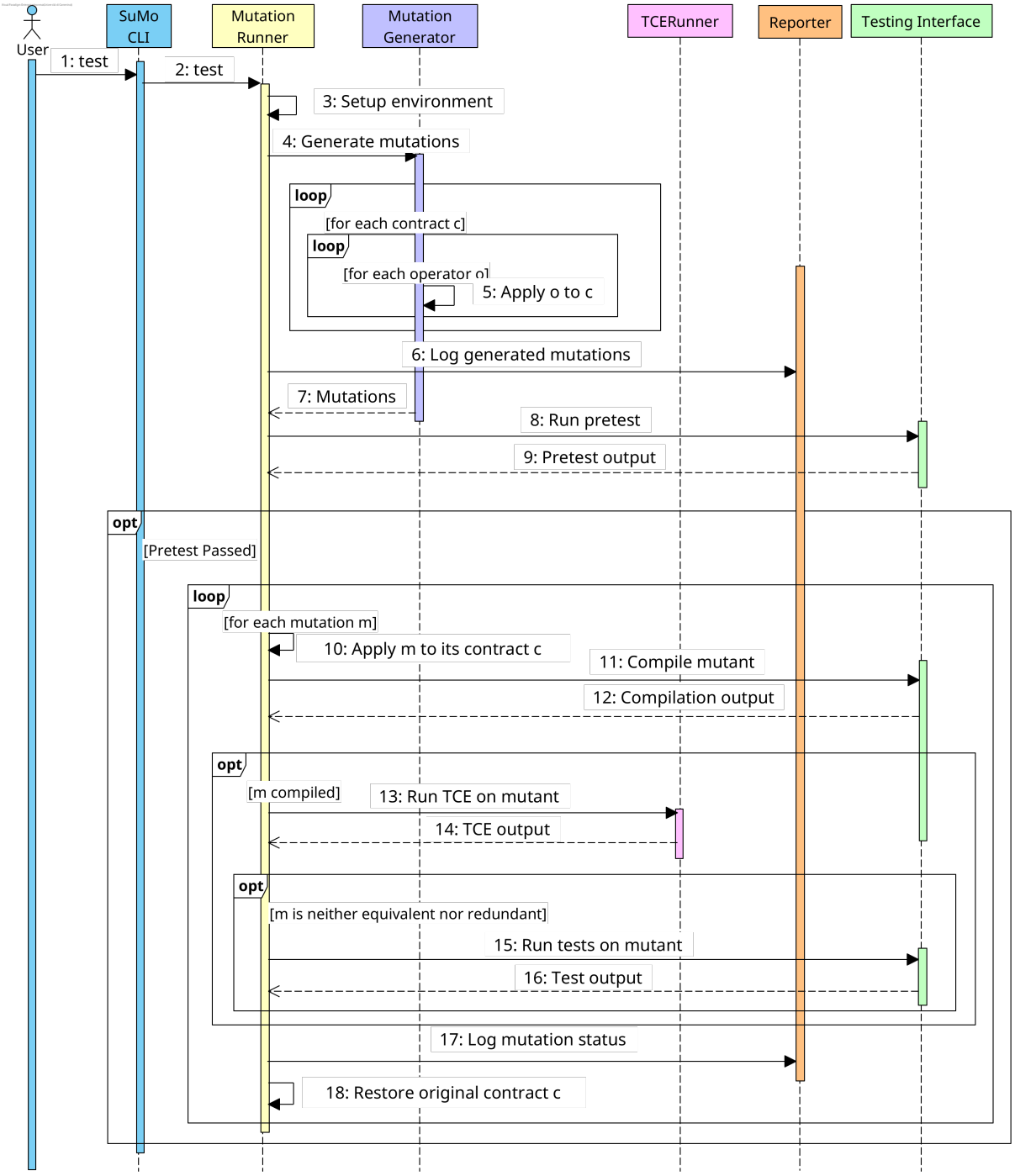


Figure 6.3: Sequence Diagram of SuMo for the Mutation Testing Process

- The mutation runner requests the mutation generator to create mutations. To this

end, the mutation generator instantiates a set of mutation operators. Each enabled operator traverses the AST of the target smart contracts, creating mutations based on its rules.

- The mutation runner logs the generated mutations to JSON via the reporter.
- The mutation runner performs a pretest to verify that the test suite passes on the original contracts in the SUT. If it doesn't, the process is terminated.

After these preliminary steps, the mutation runner has access to the set of mutations found by the generator. For each mutation, the following actions are performed:

- The mutation runner creates a mutant by applying the mutation to the source code of its contract in the SUT.
- The mutation runner attempts to compile the mutant through the testing interface. If compilation fails, the mutant status and runtime are logged in the respective mutation object.
- The mutation runner executes the TCE on the mutant using the TCERunner. If the mutant is found equivalent or redundant, it is logged, and the process for that mutant ends.
- The mutation runner starts a new test process for the mutant. Since some mutations may cause infinite loops, SuMo enforces a configurable timeout on test execution. If this limit is exceeded, the process is stopped, and the mutant is marked as timed-out. If the test completes and any test fails, the mutant is considered killed, otherwise, it remains live. As before, the respective mutation object is synchronously updated with the status and the runtime information.
- The mutation runner reverts the mutated contract in the SUT to its original state for the next mutation.

Testing Interface

The execution of compilation and testing tasks is managed by SuMo through a dedicated testing interface. When installed in the SUT, SuMo automatically detects the testing framework(s) in use by identifying relevant configuration files, such as `hardhat-config.js`. Listing 6.11 presents a minimal example of the `spawnTest()` function, which enables SuMo to initiate tests from the root directory of the SUT. Based on the detected frameworks and any optional test files specified by the user, SuMo constructs a test command that runs synchronously in the root directory of the SUT. The outcome of this command (successful, failed, or timed-out) determines the status of the testing process. If developers prefer to use a custom test script, SuMo will bypass the command-building step, instead executing the default test and compile scripts specified in the SUT's `package.json`. This flexibility allows developers to adapt SuMo's functionality to better fit their specific needs.

```
function spawnTest(testingFrameworks, testFiles) {

  //Build test command based on used testing frameworks
  let testCommand = buildTestScript(testingFrameworks, testFiles);

  //Spawn test command
  const testChild = spawnSync(testCommand, { stdio: "inherit", shell: true, cwd: rootDir,
      ↪ timeout: utils.getTestingTimeout() * 1000 });

  //Return process status code
  return testChild.status;
}

function buildTestScript(testingFrameworks, testFiles) {
  let testScript = "";
  ...
  //Build test script based on detected frameworks
  testingFrameworks.forEach(framework => {
    let testsForFramework = isOnlyFramework ? testFiles :
        ↪ utils.getTestsForFramework(framework, testFiles);
    let testCommand = "";

    switch (framework) {
      case 'hardhat':
        if (runAllTests) { testCommand = "hardhat test --bail"; }
        else { testCommand = testsForFramework.length > 0 ? "hardhat test --bail
            ↪ ${testsForFramework.join(' ')}" : ""; }
      case ...  //other frameworks are handled here
    }
    //Concat commands if needed (for hybrid test suites)
    testScript += testCommand ? (testScript ? " && " : "") + testCommand : "";
  });
  return testScript;
}
```

Listing 6.11: Spawning a Test Command through the Testing Interface

Mutation Generator

The mutation generator is responsible for the management and execution of the mutation operators. Each **mutation operator** takes in input the AST of the contract to be mutated, and it outputs a set of zero or more **mutation objects**.

Mutation Operator A mutation operator is implemented as an AST visitor. As explained earlier, the AST is obtained using `solidity-parser-antlr`, a Solidity parser built on top of a robust ANTLR4 grammar. When a mutation operator runs, it visits specific node types within the AST. If the characteristics of a node satisfy the predefined mutation preconditions, the operator generates a mutation object. Each operator may generate multiple mutations per visit, depending on the mutation opportunities in the contract under test. For example, listing 6.12 shows the implementation of the PKD operator. Its purpose is to remove the payable keyword from contract functions, so its visitor method checks each `FunctionDefinition` node. However, a node is only mutated if it meets the required preconditions: the function must have payable state mutability, and it must not be marked as virtual, override, or as the special receive Ether function. When a valid node is detected, a new mutation object is created by defining a suitable replacement. In this case, the replacement is the function signature with the payable keyword

removed. More complex operators may require multiple passes through the AST, either to mutate various node types or to gather additional semantic data. Although this extra information isn't always necessary, it can be essential for producing targeted mutations and reducing the generation of redundant or uncompilable mutants.

```
const Mutation = require('../../mutation')

function PKDOperator() {
  this.ID = "PKD";
  this.name = "payable-keyword-deletion";
}

PKDOperator.prototype.getMutations = function (file, source, visit) {
  let mutations = [];

  visit({
    FunctionDefinition: (node) => {
      if (node.stateMutability === "payable" && !node.isReceiveEther && !node.isVirtual &&
          ↪ !node.override) {

        const start = node.range[0];
        const end = node.body.range[0];
        const startLine = node.loc.start.line;
        const endLine = node.body.loc.start.line;
        const original = source.slice(start, end); //function signature
        const replace = original.replace(/payable(?![\s\S]*payable)/, "");

        mutations.push(new Mutation(file, start, end, startLine, endLine, original,
            ↪ replace, this.ID));
      }
    }
  });
  return mutations;
};

module.exports = PKDOperator;
```

Listing 6.12: Implementation of the PKD Mutation Operator

Mutation The Mutation object represents a single mutation instance applied to a smart contract during mutation testing. As shown in Listing 6.12, each Mutation object is initialized with specific properties to define its scope. Instead, certain properties are dynamically updated at run-time (e.g.: the status). These are shown in Table 6.14.

TCE Runner

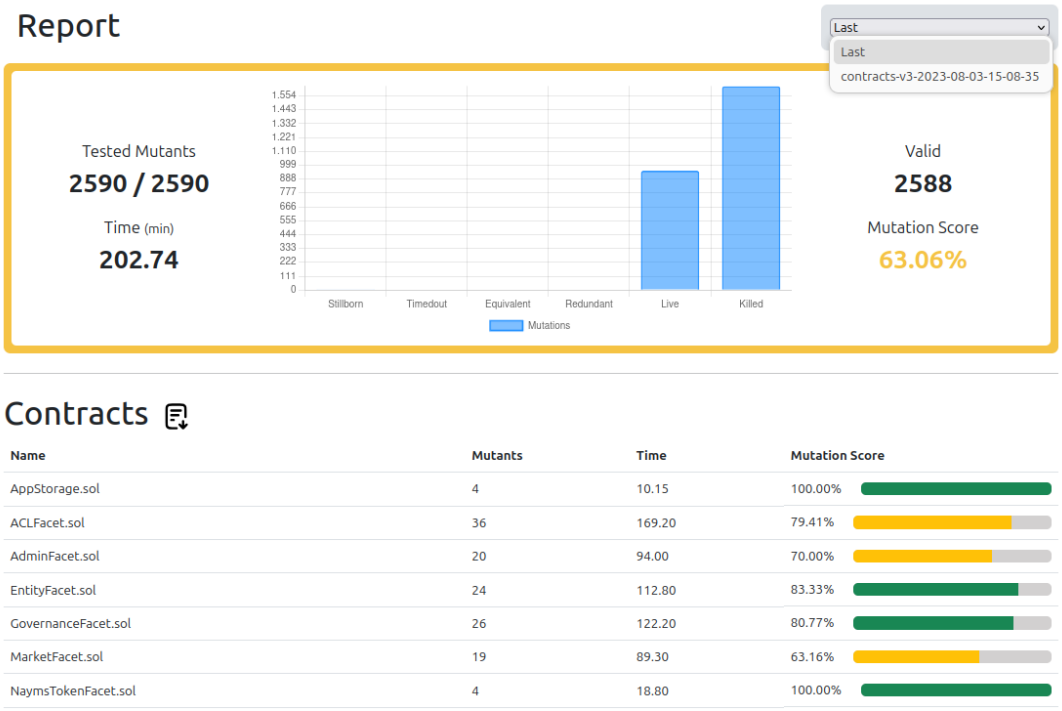
To improve the reliability and the cost-effectiveness of mutation testing, SuMo implements a module that uses the Trivial Compiler Equivalence to prune redundant and equivalent mutants. TCE operates on the assumption that bytecode optimizations by the compiler will yield identical bytecode for functionally equivalent contracts. Thus, the TCE Runner checks each mutant's bytecode against the original and previously tested mutants: mutants matching their original contract's bytecode are marked as *equivalent*, while mutants identical to earlier ones are labeled *redundant*. Mutants with unique bytecodes are stored in a mutation map for tracking.

Property	Description	Type	Example
`ID`	The ID of the mutation, generated by applying a hash function to some of its other properties.	string	`"m72e68d82"`
`file`	The path to the contract file in the SUT from which the mutation is generated.	string	`".../Rain.sol"`
`start`	The character position within the file where the mutation begins.	number	1258
`end`	The character position within the file where the mutation ends.	number	1266
`startLine`	The start line number for the mutation's location.	number	58
`endLine`	The end line number for the mutation's location.	number	58
`original`	The code segment to be mutated	string	`"isActive"`
`replace`	The mutated version of the original code segment.	string	`"false"`
`operator`	The ID of the mutation operator used to produce the mutation.	string	`"CSC"`
`status`	The mutant's outcome during the mutation testing process.	string	`"killed"`
`testingTime`	The time taken to test the mutant.	number	6258

Table 6.14: Properties of the Mutation Object

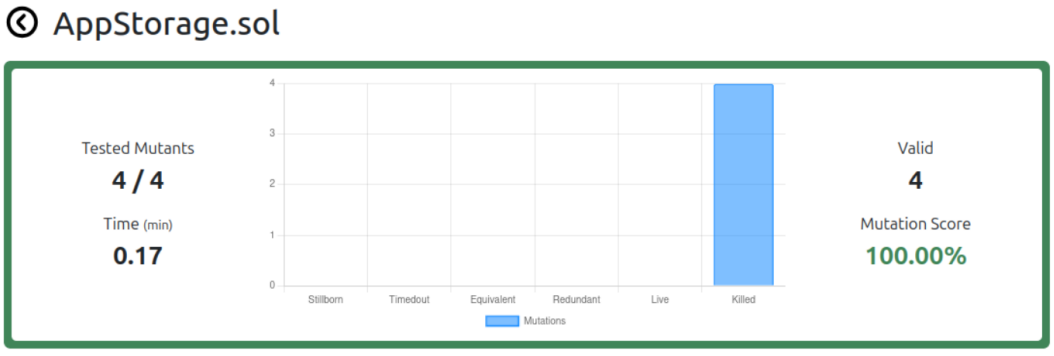
Reporter

The reporter logs key information about the ongoing mutation testing process. Each time the mutation runner tests a new mutant, the reporter updates its status and testing time in the JSON mutations list. This file is updated synchronously with each processed mutant so that users can dynamically monitor progress. To enhance result visualization, SuMo also generates an HTML report implemented as a multi-page application. The report has two main sections: the home page and the contract page. As shown in Figure 6.4, the report's main page presents a summary of each contract file subjected to mutation testing. For each contract, users see an overview of tested mutants, testing duration, and mutation score. Additionally, a table on the home page organizes data by mutation operators (not shown in Figure 6.4), allowing users to analyze each operator in detail. From the home page it is also possible to inspect the full *history* of SuMo runs executed on the same SUT. Users can choose a specific run via a drop-down menu in the upper-right corner, and they can export mutant and operator table data to CSV. This allows users to easily monitor how the mutation score changes as they refine the test suite. Clicking on a contract opens the contract page, which provides specific details for that contract. As shown in Figure 6.5, the contract page includes a summary of data for the selected contract and a list of generated mutants. Users can filter this list by operator or status, and view the code diff between the original and mutated versions by clicking on a specific mutant row. Finally, the contract also allows the mutant list to be exported to CSV via a dedicated button.



Name	Mutants	Time	Mutation Score
AppStorage.sol	4	10.15	100.00%
ACLFacet.sol	36	169.20	79.41%
AdminFacet.sol	20	94.00	70.00%
EntityFacet.sol	24	112.80	83.33%
GovernanceFacet.sol	26	122.20	80.77%
MarketFacet.sol	19	89.30	63.16%
NaymsTokenFacet.sol	4	18.80	100.00%

Figure 6.4: Home Page of the SuMo Test Report



Hash	Operator	Start Line	End Line	Original	Replacement	Status	Time(s)
m10abfd14	DLO	87	87	AppStorage storage ds	AppStorage memory ds	killed	2.79
ma3e5929f	DLO	94	94	FunctionLockedStorage storage ds	FunctionLockedStorage memory ds	killed	1.79
mccd798a1	MCR	84	84	keccak256	sha256	killed	1.78
mbda5e84e	MCR	85	85	keccak256	sha256	killed	3.79

Figure 6.5: Contract Page of the SuMo Test Report

SuMo Configuration

When SuMo is installed as a dependency, it automatically creates a "`sumo-config.js`" file in the root directory of the SUT. This file allows the user to specify several options before starting the mutation testing process. Listing 6.13 shows an example of the configuration file. The `buildDir`, `contractsDir` and the `testDir` indicate the name of the directory within the SUT containing the compilation artifacts, contract files to be mutated, and test files to be evaluated. These are set by default to the conventional directory names used in most Solidity projects. The `skipContracts` and `skipTests` arrays permit to specify a list of contracts and test files to be ignored by SuMo. The `minimal` and the `tce` flags allows the tester to enable and disable the operator optimizations and the Trivial Compiler Equivalence. The `testingFramework` and *network* fields are used to set the testing framework and (optionally) the usage of Ganache during testing. Lastly, the *testingTimeOutInSec* permits to tweak after how much time a mutant is considered timed-out during testing. Additional details about the configuration options can be found in the SuMo repository.

```
module.exports = {
  buildDir: 'build',
  contractsDir: 'contracts',
  testDir: 'test',
  skipContracts: ['contractName.sol'],
  skipTests: ['testFileName.js'],
  testingTimeOutInSec: 300,
  network: "none",
  testingFramework: "hardhat",
  minimal: false,
  tce: false
}
```

Listing 6.13: Example of sumo-config.js

6.5.4 Adoption and Technical Impact

To gauge the real-world relevance of the tool, we examined its usage and adoption within the software and research community. The tool is hosted on GitHub and, as of January 2025, has accumulated approximately 80 stars and several forks and pull requests, indicating community interest and engagement. Furthermore, it has been incorporated into ten publicly available Solidity projects for the assessment of test suites.

6.6 Experiment Methodology

In the following, we present the experiment subjects and the methodology we followed to answer the Research Questions posed in Section 6.3.

6.6.1 Methodology for RQ1

To address RQ1, we run SuMo on a set of Solidity projects, applying the complete set of Non-Optimized mutation operators. Figure 6.6 depicts the steps we followed to run SuMo on the subjects. Each mutation testing run comprises the following steps:

1. **Mutant Generation:** We mutate the original set of contracts I with the Non-Optimized operators to generate a set of mutants M_n.

2. **Compilation:** SuMo compiles each mutant $m \in M_n$ and each contract $i \in I$. This produces two sets of binaries: C_i for the original contracts, and C_n for the mutants. At this stage, the set of stillborn mutants S_n that do not compile are also discarded from M_n.

3. **Trivial Compiler Equivalence:** SuMo applies the TCE to detect and discard (trivially) equivalent and redundant mutants. Specifically, the Equivalent Mutant Detector (Introduced in Section 6.5.3) compares each mutant binary in C_n against the original contract's in C_i to detect equivalencies. It also compares the mutant binaries in C_n against each other to detect redundancies. Then, it outputs two different artifacts: D_n is the set of mutants discarded with the TCE, while the set $T_n = C_n - D_n$ contains the mutants under test that must be run by SuMo.

4. **Mutation Testing:** We test each mutant $m \in T_n$ with SuMo. Depending on the outcome of the testing process, m can either be marked as killed or live. The subset of killed mutants K_n permits to calculate the mutation score MS_n achieved by the test suite.

After running mutation testing, we evaluate the Mutation Score achieved by the test suites with respect to each operator, and we inspect their live mutants to determine their significance. As introduced in Section 6.3, this analysis will mostly focus on those operators that target Solidity-specific programming aspects, as well as the novel operators introduced by SuMo.

6.6.2 Methodology for RQ2

To address RQ2, we repeat the mutation testing process shown in Figure 6.6 using the optimized mutant set M_o. The goal is to evaluate the impact of optimizations in reducing the number of equivalent, redundant, and subsumed mutants generated, and whether these optimizations achieve efficiency without compromising the quality and depth of mutation testing. To assess the benefits and potential drawbacks of these optimizations, we consider the following metrics:

1 - Time Costs: This metric measures the time spent in the *mutation generation* and *testing* phases. By reducing invalid and unhelpful mutants, the testing process becomes faster, thus making it more scalable. However, if the optimizations lead to significant information loss (fewer valuable mutants), the reduced time cost may not justify their use.

2 - Information Loss: This metric captures how many live mutants, which survive all tests, are uniquely generated by the non-optimized set of operators. Therefore, it

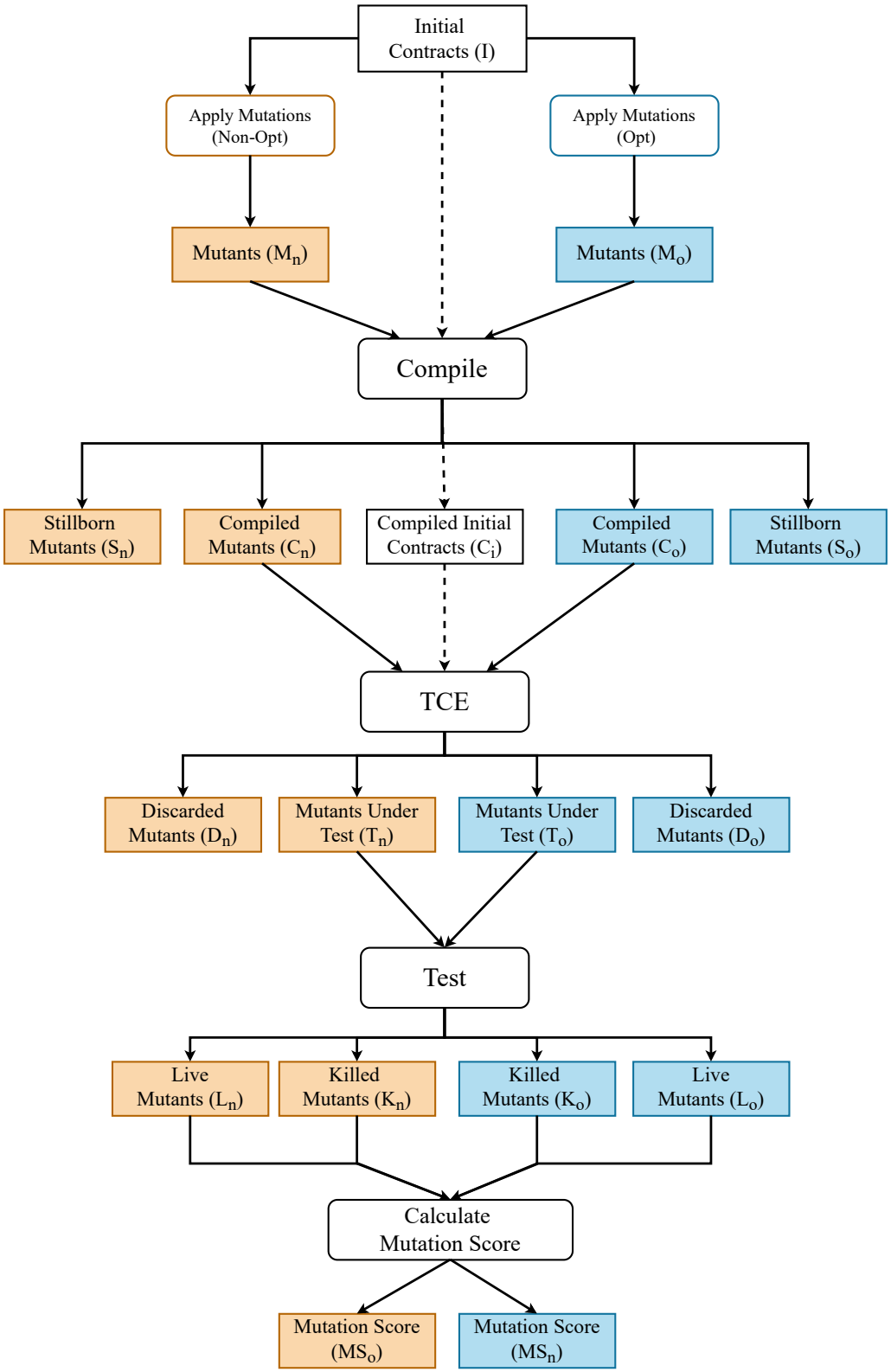


Figure 6.6: Mutation Testing Process for the Experiment

measures the extent to which the optimized set fails to generate mutants that could otherwise be useful for identifying gaps in the test suite.

3 - Subsumed Mutants: Identifying subsumed mutants is essential because generating such mutants does not provide additional value in testing but increases the time cost and complexity. Optimized operators aim to reduce the number of subsumed mutants generated, thus improving efficiency. We follow the definition by Kurtz et al. [94], where mutant $m1$ is subsumed by mutant $m2$ if every test that kills $m2$ also kills $m1$. Since computing true subsumption requires testing against all possible test sets, we approximate subsumption relationships using the specific test suite for each project. To do this:

1. We save the mutation testing results as a **mutation matrix**, which contains the outcome of each test case against each mutant;
2. We compute the **mutant subsumption relationships** among the mutants generated by the same mutation operator;
3. We add the specific replacement rule that generated the mutants to each subsumption relationship;
4. We determine if the subsumption relationships established among any two replacement rules always held for the considered project by searching for a counterexample (i.e., a non-subsumption relationship);
5. We build the **mutant subsumption graph** containing all the subsumption relationships that always hold for the considered project, and we used it to compute the subsumed mutants.

6.6.3 Methodology for RQ3

To address RQ3, we analyze whether live mutants provide actionable insights for improving the test suite. For this, we select one of the Solidity projects used in the earlier experiments, we review the live mutants (L_o) generated from the Optimized mutation operators. Then, we attempt to improve the test suite by designing new test cases or refining existing ones based on these mutants. After the test suite is improved, we re-run SuMo to calculate the new Mutation Score and evaluate whether the live mutants led to a meaningful enhancement in the test suite's fault-detection capabilities. Based on these results, we discuss mutants that proved particularly valuable in highlighting weaknesses or generating new test cases. Given the effort of manually analyzing live mutants, we limit this analysis to a single Solidity project. This manual analysis process is time-intensive and relies on the tester's domain knowledge of the specific project, making it impractical to extend to multiple projects.

6.6.4 Experiment Subjects

To conduct the experiments, we selected several open-source Ethereum projects from GitHub. We chose applications showing some complexity and from different domains to mutate a wider range of Solidity features. At the same time, we looked for projects with

Name	#C	CLOC	#T	TLOC	Stmt Cov.	Branch Cov.
Alice[10]	28	970	201	1804	97,16	70,69
Blackjack[11]	1	369	13	98	70,75	56,25
Coinosis[12]	2	99	28	819	100	87,5
EtherCrowdfunding[13]	1	428	35	902	93,83	71,3
Safe Contractss[14]	21	1422	258	3531	95,23	99,73
Bionic Event[15]	2	182	25	261	90	65

Table 6.15: Subject DApps Metrics

a high-coverage test suite to ensure some investment in testing. Table 6.15 shows the metrics for the selected applications, including the total number of contract files (#C) with their lines of code (CLOC), the total number of implemented test methods (#T) with their lines of code (TLOC) and the relative Statement and Branch Coverage.

To answer RQ1 and RQ2, we considered the following projects:

1. *Alice* is a donation platform built on Ethereum;
2. *Blackjack* is a DApp that permits to run a Blackjack game on Ethereum;
3. *Coinosis* is a collaborative learning DApp that promotes the creation of study groups on various topics. People with common interests can join the group, and be rewarded for sharing their knowledge based on the recognition of the participants;
4. *EtherCrowdfunding* is a DApp for the organization of crowdfunding campaigns. It supports the creation of a new campaign, the donation of funds from the donors, and the reception of funds from the beneficiaries;
5. *Safe Contract* is a smart contract wallet with multi-signature functionality that allows secure management of blockchain assets.

To answer RQ3, we selected the *Bionic Event* project, as it offers a good balance between test suite complexity and number of generated mutants. Bionic Event is a DApp that supports the creation and management of events, as well as the purchase, validation, and transfer of tickets.

6.7 Discussion of Results

Here we provide a general discussion of the experimental results, whereas in the next sections we answer the research questions RQ1 (Section 6.7.1), RQ2 (Section 6.7.2) and RQ3 (Section 6.7.3). Table 6.16(1) and Table 6.16(2) provide the results for the mutation testing run carried out with the Non-Optimized and the Optimized operators, respectively. For each DApp, the two tables show the total number of generated mutants (M_n and M_o), stillborn mutants (S_n and S_o), the compiled mutants (C_n and C_o), the mutants discarded with the Trivial Compiler Equivalence (D_n and D_o), the mutants under test

(T_n and T_o), the killed mutants (K_n and K_o), and the live mutants (L_n and L_o). The last column shows the Mutation Score achieved by the relative test suite. Moreover, for each DApp the first line reports the data when all the mutation operators were used, the second line just focuses on Solidity-related operators, while the third one reports the observed values for the new operators introduced in SuMo.

Test Suite Adequacy As can be seen from Table 6.16, the test suites shipped with the selected applications were unable to kill a considerable amount of mutants, both during the Non-Optimized and the Optimized run. The test suites achieved a global MS_n = 61,9% and MS_o = 61,5%., and the adequacy further decreases if we only consider Solidity-specific operators.

Not surprisingly, Solidity-specific mutants seem more difficult to kill and this could be due to the fact that programmers need to get more acquainted with the novel language mechanisms. Furthermore, ≈ 10% of the Non-Optimized live mutants, and ≈ 6% of the Optimized ones, were generated by the SuMo-specific operators.

Figure 6.7 compares the adequacy value of each test suite in terms of its 1) Statement Coverage, 2) Branch Coverage, 3) Mutation Score (Optimized), and 4) Mutation Score (Non-Optimized). As can be seen, the Mutation Score and the code coverage values convey a different perception of the test suites' quality. The Mutation Score is consistently lower for each selected case study. Particularly interesting is the case of Safe-Contracts that, notwithstanding a ≈ 100% statement and branch coverage, only reached a Mutation Score of ≈ 76% during both mutation runs.

Overall, these results confirm that test suites with high values in the assessment of structural coverage parameters do not necessarily ensure the reliability of the smart contract code, as they can still miss a large number of faults. A detailed discussion on the effects of each mutation operator is reported in section 6.7.1.

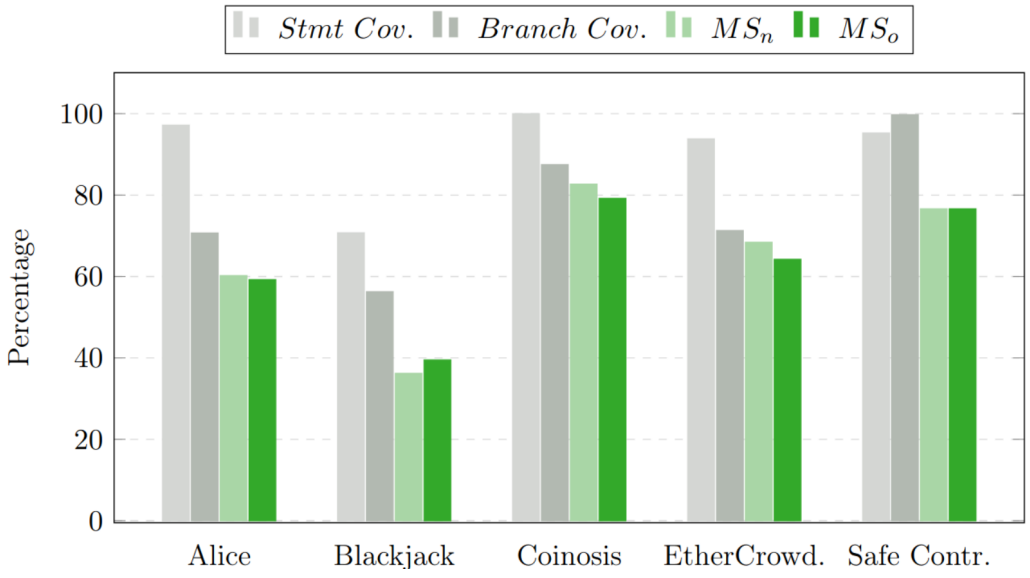


Figure 6.7: Adequacy of the test suites for the selected DApps

Table 6.16: Experimental Results

(1) Non-optimized

DApp	**Mutation Operators**	M_n	S_n	C_n	D_n	T_n	K_n	L_n	$MS_n(\%)$
`Alice`	All	1296	153	1143	68	1075	647	428	60,2
	Solidity	799	151	648	54	594	334	260	56,2
	SuMo	56	15	41	0	41	20	21	48,8
`Blackjack`	All	877	0	877	74	803	291	513	36,2
	Solidity	265	0	265	36	229	94	136	41
	SuMo	159	0	159	3	156	77	79	49,4
`Coinosis`	All	231	0	231	29	202	167	35	82,7
	Solidity	106	0	106	24	82	63	19	76,8
	SuMo	26	0	26	0	26	26	0	100
`Ether Crowdfund.`	All	899	2	897	108	789	540	249	68,4
	Solidity	425	0	425	68	357	206	151	57,7
	SuMo	140	0	140	1	139	129	10	92,8
`Safe Contracts`	All	1424	135	1289	376	913	699	214	76,6
	Solidity	626	97	529	169	360	268	92	74,4
	SuMo	130	17	113	12	101	71	30	70,3
Total	All	4727	290	4437	655	3782	2343	1439	**61,9**

(2) Optimized

DApp	**Mutation Operators**	M_o	S_o	C_o	D_o	T_o	K_o	L_o	$MS_o(\%)$
`Alice`	All	1036	147	889	59	830	491	339	59,2
	Solidity	775	145	630	54	576	328	248	56,9
	SuMo	32	9	23	0	23	14	9	60,9
`Blackjack`	All	536	0	536	56	480	189	291	39,6
	Solidity	213	0	213	36	177	76	101	43,5
	SuMo	65	0	65	1	64	34	30	53,1
`Coinosis`	All	156	0	156	26	130	103	27	79,2
	Solidity	94	0	94	24	70	51	19	72,8
	SuMo	14	0	14	0	14	14	0	100
`Ether Crowdfund.`	All	600	0	600	97	503	323	180	64,2
	Solidity	370	0	370	68	302	151	151	50
	SuMo	70	0	70	1	69	60	9	86,9
`Safe Contracts`	All	971	127	844	259	585	448	137	76,6
	Solidity	557	89	468	164	304	229	75	75,3
	SuMo	61	9	52	7	45	32	13	71,1
Total	All	3299	274	3025	497	2528	1554	974	**61,5**

Legend: M: total mutants, S: stillborn mutants, C: compiled mutants, D: equivalent and redundant mutants discarded by the TCE, T: mutants under test, K: killed mutants, L: live mutants, MS: mutation score

Stillborn Mutants The percentage of stillborn mutants generated by SuMo is relatively small, which suggests the effectiveness of the implemented precondition checks. In particular, we can observe that only 8,3% of the Optimized mutants and 6,13% of the Non-Optimized mutants were marked as stillborn.

Compilation errors are unevenly distributed among the different types of operators, and they seem to be more frequent when dealing with Solidity-specific mutations. As we better discuss in next section, we pinpointed three mutation operators that play a major role in the generation of stillborn mutants. CCD and ETR feature the highest stillborn mutant generation rate, while the largest amount of stillborn mutants was produced by FVR. This information can help testers to adapt the mutation testing process to their needs, so to reduce the costs associated with the compilation phase.

Limiting the generation of stillborn mutants can make a difference in the long run. Table 6.17 reports the average time wasted by a single stillborn mutant on 20 independent runs (Col. 2) for each DApp. Such values include the time required for applying the mutation to the contract and the time wasted for attempting its compilation. The data shows that the average slowdown induced by a single stillborn mutant is around 7 seconds, which can vary depending on the characteristics of the contract under test.

DApp	**Avg. Time Wasted (s)**
`Alice`	7.59
`Coinosis`	5.92
`Blackjack`	6.00
`EtherCrowdfunding`	8.90
`Safe Contracts`	7.20
Avg.	7.12

Table 6.17: Average Time Overhead of a Stillborn Mutant

Equivalent and Redundant Mutants Column 5 shows that TCE detected a significant amount of program equivalencies both during the Optimized and the Non-Optimized test runs. In particular, $|D_n|$ = 655 and $|D_o|$ = 497. This means that 13,8% of the mutants in M_n and 15% of the mutants in M_o were marked as either equivalent or redundant. Thus, applying TCE contributed to significantly decreasing the cost of the testing process, and obtaining a more reliable Mutation Score.

Surprisingly, most equivalencies were detected among the Solidity-specific mutants. Particularly interesting here is the case of the Safe-Contracts DApp. Indeed, almost 30% of its Solidity mutants were discarded before testing. The SuMo-specific operators, on the other hand, were seldom marked as equivalent or redundant.

6.7.1 Impact of the Mutation Operators

In this section we discuss the results achieved by mutation operators to answer RQ1: "*Do the SuMo-specific and Solidity-specific operators contribute to the identification of*

weaknesses in Smart Contract test suites?".

As shown in Figure 6.9 (for the Non-Optimized version) and 6.8 (for the Optimized version), the Solidity-Specific operators contribute with a relevant fraction to the total number of live mutants. With the exception of the Blackjack project, Solidity mutations consistently produced more live mutants compared to the general ones.

During the Optimized runs, the Solidity-specific operators achieved an average Mutation Score of 58.4%, whereas the general operators scored 65.4%. A similar pattern emerged in the Non-Optimized runs, where Solidity-specific operators had a Mutation Score of 59.4%, compared to 63.8% for the general operators.

The Mutation Score of the SuMo-specific operators was generally higher throughout the experiments, with the exception of the Safe-Contracts project. The average SuMo-specific Mutation Score is 69, 8% for the Non-Optimized runs, and 71, 6% for the Optimized runs. The SuMo-specific mutations do not seem more difficult to kill, due to the fact that they target both traditional and Solidity-specific programming aspects.

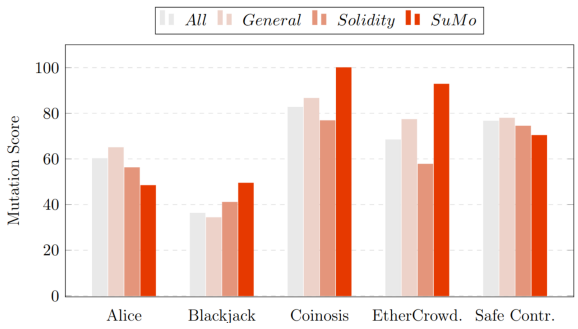


Figure 6.8: Mutation Score achieved by the projects (Non-Optimized)

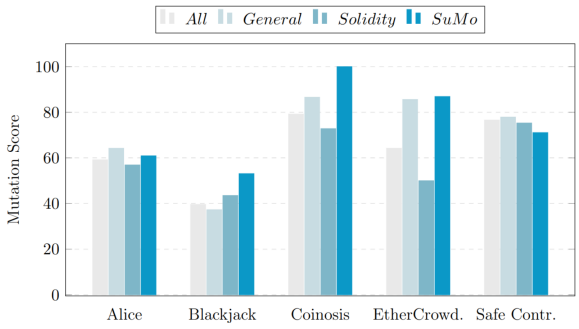


Figure 6.9: Mutation Score achieved by the projects (Optimized)

Tables 6.18 and 6.19 provide details of the results for the Non-Optimized and Optimized mutation operators, respectively. For each operator, the tables describe the total number of generated mutants (M), the number of stillborn mutants (S), the number of mutants discarded with the Trivial Compiler Equivalence (D), the number of mutants under test (T) and the achieved Mutation Score (MS). The experimental results reveal which types of injected faults are more likely to escape detection by existing test suites.

While it is not possible to conduct an exhaustive analysis of each live mutant, we observed distinct patterns in the survival rates and effectiveness of each operator, particularly those targeting Solidity-specific constructs. The results show that the novel and Solidity-specific operators, including AVR, ECS, and EED, significantly contribute to the identification of gaps in smart contract test suites. They reveal areas that are commonly overlooked by testers, such as address manipulation, event emission, and exception handling. These are gaps that could lead to bugs in real-world deployments. The test suites were generally more effective at handling traditional faults simulated by general operators (e.g., mathematical functions) than Solidity-specific or SuMo-specific faults. This reinforces the importance of using specialized mutation operators to thoroughly test smart contracts, given the unique characteristics of Solidity and the blockchain environment.

AVR - Address Value Replacement The AVR mutants were often killed by the provided test suites. The effects of an address mutation on the contract execution can

Table 6.18: Results for the Non-Optimized Mutation Operators

ID	Mutation Operator	M_n	S_n	D_n	T_n	MS(%)
ACM	**Argument Change of overloaded Method call**	-	-	-	-	-
AOR	Assignment Operator Replacement	25	0	2	23	52,2
AVR	Address Value Replacement	162	6	26	130	80
BCRD	Break and Continue Replacement and Deletion	6	0	1	5	60
BLR	Boolean Literal Replacement	18	0	0	18	66,7
BOR	Binary Operator Replacement	1637	2	214	1421	66,8
CBD	Catch Block Deletion	-	-	-	-	-
CCD	**Contract Constructor Deletion**	26	10	0	16	93,7
CSC	Conditional Statement Change	149	0	27	122	61,5
DLR	Data Location keyword Replacement	121	66	4	51	72,5
DOD	Delete Operator Deletion	4	0	3	1	0
ECS	**Explicit Conversion to Smaller type**	22	7	2	13	69,2
EED	Event Emission Deletion	60	4	1	55	34,5
EHC	Exception Handling statement Change	314	1	24	289	40,1
ER	**Enum Replacement**	96	0	4	92	71,8
ETR	Ether Transfer function Replacement	30	22	0	8	0
FVR	Function Visibility Replacement	556	111	203	242	66,9
GVR	**Global Variable Replacement**	239	13	6	220	64,5
HLR	Hexadecimal Literal Replacement	-	-	-	-	-
ICM	Increments Mirror	1	0	0	1	100
ILR	Integer Literal Replacement	346	26	34	286	50,3
LSC	Loop Statement Change	40	0	5	35	97,1
MCR	**Mathematical and Cryptographic function Replacement**	29	1	2	26	92,3
MOC	**Modifiers Order Change**	-	-	-	-	-
MOD	Modifier Deletion	71	0	2	69	37,7
MOI	Modifier Insertion	60	1	4	55	61,8
MOR	Modifier Replacement	36	0	0	36	61
OLFD	**Overloaded Function Deletion**	10	1	1	8	75
OMD	**Overridden Modifier Deletion**	-	-	-	-	-
ORFD	Overridden Function Deletion	11	6	3	2	0
PKD	Payable Keyword Deletion	13	0	0	13	100
RSD	Return Statement Deletion	59	0	2	57	91,2
RVS	**Return Values Swap**	5	0	1	4	100
SCEC	Switch Call Expression Casting	-	-	-	-	-
SFD	Selfdestruct Function Deletion	3	0	0	3	66,7
SFI	Selfdestruct Function Insertion	3	0	1	2	100
SFR	**SafeMath Function Replacement**	84	0	0	84	67,8
SKD	Super Keyword Deletion	5	0	0	5	60
SKI	Super Keyword Insertion	-	-	-	-	-
SLR	String Literal Replacement	83	0	9	74	32,4
TOR	Transaction Origin Replacement	108	1	2	105	15,2
UORD	Unary Operator Replacement Deletion	57	0	2	55	74,5
VUR	Variable Unit Replacement	-	-	-	-	-
VVR	Variable Visibility Replacement	238	12	70	156	75

Table 6.19: Results for the Optimized Mutation Operators

ID	Mutation Operator	M_o	S_o	D_o	T_o	MS (%)
AOR	Assignment Operator Replacement	10	0	0	10	60
BOR	Binary Operator Replacement	493	0	65	428	78,3
ER	Enum Replacement	39	0	2	37	70,3
GVR	Global Variable Replacement	103	0	2	101	66,3
RVS	Return Values Swap	5	0	1	4	100
SFR	SafeMath Function Replacement	21	0	0	21	66,7
VUR	Variable Unit Replacement	0	-	-	-	-

Legend:
M_o:total mutants, S_o: stillborn mutants,
D_o: equivalent and redundant mutants discarded by the TCE,
T_o: tested mutants, MS: mutation score

vary largely. For instance, performing an incorrect address assignment might lead to the retrieval of the wrong balance value. In any case, altering an address is likely to have a drastic effect on the behavior of the contract. Our analysis of the live mutants suggests that the AVR operator can expose a wide range of issues with a given test suite. For instance, an AVR survivor might indicate that the contract state is not properly checked after an address assignment. Several AVR survivors were also useful to spot a lack of coverage for entire methods as well as guard clauses that include conditions on addresses.

CCD - Contract Constructor Deletion The novel CCD operator generated 16 mutants under test, most of which were killed by the tests. This mutation overrides the user-defined constructor and prevents the initialization operations of the contract from taking place. The high Mutation Score achieved by CCD indicates that the testers often pay attention to the correctness of the constructor. Even though this type of mutation is not very subtle, our results still include some live mutants. Furthermore, CCD can only generate one mutant per contract, meaning that its execution constitutes a very small overhead in the scope of a complete mutation testing campaign.

DLR - Data Location Replacement The DLR operator generated 51 mutants under test, some of which survived testing. We observed that most surviving mutations replaced the `storage` keyword with the `memory` keyword. This can prevent any modification to the affected variable from being saved to state. On the other hand, the mutations that swap `memory` with `storage` can cause unwanted variable overwriting. These mutants can survive if the effects of the mutation do not propagate through the program. However, it is also possible that the test suite does not properly check the contract state after the transaction.

DOD - Delete Operator Deletion The DOD operator only generated 1 mutant under test. This is not surprising as the `delete` statement is not as frequent as other operators

in real-world Smart Contracts. The survival of this mutant shows a weakness in the test suite; the state of the contract is not properly checked after the affected variable is set to its default value.

ECS - Explicit Conversion to Smaller type The novel ECS operator only generated 13 mutants under test. Indeed, explicit conversions are not extremely frequent in smart contract development. During our experiments, several ECS mutants survived testing. An ECS mutant is more likely to be killed whenever the fault can easily propagate and cause anomalous contract behavior. For instance, altering an explicit conversion might drastically change the value that is being assigned to a constant state variable. However, in some cases, the effects of the mutation might be more subtle and difficult to detect for a test. For instance, ECS might alter the value of an address that is supposed to receive funds. Since the mutated address is likely to be orphaned, the ether will be lost during transfer. Therefore, this operator can be useful to improve the reliability of the contract code.

EED - Event Emission Deletion The EED mutations were regularly missed by the tests as well. An EED mutant can survive when the affected event is unreachable or difficult to trigger during testing. However, the selected case studies feature events that appear to be reachable from the test suite. This indicates that events are often overlooked during testing. The EED operator encourages testers to take advantage of the event logging mechanism. In fact, the emission of an event (or lack thereof) can be extremely useful to ensure the correct behavior of the contract.

EHC - Exception Handling Change The number of generated EHC mutants is quite high in relation to most classes of mutants. In fact, `assert`, `require` and `revert` are prevailing statements throughout a typical Solidity project. The overall Mutation Score achieved by EHC is very low. Many of the mutants that were marked as live feature a commented-out exception-handling statement. This result suggests a general lack of tests that attempt to make the condition check fail. Indeed, developers often neglect unhappy paths. This cannot ensure the proper functioning of the error handling mechanisms in the contract.

ER - Enum Replacement The novel ER operator achieved a moderately low Mutation Score. Around 30% of the generated mutants survived testing, both with and without optimizations. The surviving mutations targeted enum values in different contexts, such as return statements or the initialization of state variables. Even though the ER operator features a higher Mutation Score than other classes, it can still provide useful insights about areas of code that need more thorough testing.

ETR - Ether Transfer Replacement The ETR operator generated 8 mutants under test, none of which were killed by the tests. This result is not surprising because swapping ether transfer functions does not always affect the contract execution. In fact, the main difference between `send`, `transfer` and `call` is how they behave upon error. Given our results, it is safe to assume that testers rarely simulate a faulty ether transfer. Indeed,

designing dedicated test scenarios that force each transfer operation to fail might not always be feasible, as it can require too much effort. However, considering that a Solidity project typically contains a small number of transfer operations, the usage of ETR can be considered.

FVR and VVR - Function and Variable Visibility Replacement The FVR (Function Visibility Replacement) and VVR (Variable Visibility Replacement) mutants were sometimes missed by the tests. While VVR mostly generated testable mutants, the FVR operator caused a significant amount of compilation errors. Indeed, FVR swaps each function visibility keyword with three different keywords, leading to the generation of a copious amount of mutants. In some cases, mutating the visibility keyword can generate a stillborn mutant. For instance, mutating a function from `private` to `external` can prevent the target contract from calling the function, thus causing a compilation error. Most FVR and VVR survivors contain a looser visibility keyword than the original contract. This result was predictable, as testers rarely check whether unauthorized accounts can call restricted functions. Mutations that swap a visibility keyword with a more restrictive one are also likely to be missed, especially if the semantic difference with the original contract is very subtle. These results suggest that the provided test suites do not always directly test variable and function visibility. Indeed, defining dedicated test cases requires additional effort. Analyzing the survivors challenges the developer to understand whether the contract code is associated with the correct visibility keywords.

GVR - Global Variable Replacement More than half of the mutants generated by the novel GVR operator was killed by the tests. Although both the Optimized and the Non-Optimized operators achieved average mutation scores, the latter is responsible for a higher percentage of live mutants. This is likely caused by the diverse array of mutations injected by the extended rules. Most mutations targeted the `block.timestamp` (or `now`) and the `msg.value` variables, which are widely used in a typical smart contract project. The results suggest that faults in the usage of global variables can be subtle and difficult to detect, which can help to design improved test data. Therefore, we believe that GVR represents a useful addition to our set.

MCR - Mathematical and Cryptographic function Replacement The novel MCR operator generated 26 mutants under test, most of which were killed. Our analysis revealed that each survivor contained a mutated cryptographic function. Despite being quite evident, this fault was not detected on several occasions, which confirms the usefulness of the novel operator.

MOD - Modifier Operator Deletion The mutants generated by the MOD operator were rarely killed throughout our experiments. The modifiers deleted by MOD mostly implemented precondition checks. This further confirms a lack of tests that exercise the unhappy path, such as calling a function from an unauthorized account or performing an operation at the wrong stage.

MOI and MOR - Modifier Operator Insertion and Replacement The Mutation Score achieved by MOI (Modifier Operator Insertion) is in line with MOR (Modifier Operator Replacement); both of them achieve a higher Mutation Score than MOD, but many mutations were still missed by the tests. Indeed, replacing or attaching a modifier to a function is a less subtle fault that can prevent a test transaction from succeeding. Nevertheless, a MOI or MOR mutant can survive if the tests manage to trigger the "true" branch of the added modifier. For instance, the transaction might be issued from an account that passes the authorization check. It is also possible that the tests never attempt to call the mutated function.

OLFD - Overloaded Function Deletion The novel OLFD operator produced 8 mutants under test, some of which survived testing. Even though the sample of mutants is limited, we believe that targeting overloaded functions can be a useful addition to the set of possible mutations. When the test suite cannot detect an OLFD mutant, there might be errors in the contract that prevent the overloaded method from being called. The most likely reason, however, is that the deleted method is never actually called by the tests. OLFD encourages the developers to thoroughly check the code for the presence of faults and trivial methods and to test all the possible method versions.

PKD - Payable Keyword Deletion The PKD operator generated 13 mutants under test, none of which survived testing. Only those tests that send Ether to the mutated function can kill a PKD mutant. The high Mutation Score achieved by PKD confirms developers test business-critical functions to ensure the correct reception of funds.

RSD and RVS - Return Statement Deletion and Values Swap The RSD (Return Statement Deletion) operator generated 57 mutants under test, most of which were detected by the tests. The Mutation Score achieved by the novel RVS (Return Values Swap) operator is high as well. This is not surprising given the fairly evident type of fault injected into the code. Nevertheless, the few instances of live mutants suggest that testers do not always pay attention to the correctness of the return values, especially in the case of multiple branches.

SFD and SFI - Selfdestruct Function Deletion and Insertion The SFD (Selfdestruct Function Deletion) and SFI (Selfdestruct Function Insertion) operators show that the self-destruct mechanism is sometimes overlooked by the testers. In particular, the live SFD mutant indicates that the provided test suite never attempts to destroy the contract. This is a dangerous omission, as a flawed self-destruct implementation can prevent the elimination of a contract from the blockchain. SFI mutations on the other hand are easier to detect, as they can trigger unexpected self-destruct invocations.

SFR - SafeMath Function Replacement The results of the novel SFR operator show that arithmetic faults caused by the incorrect usage of the SafeMath library are not always caught by the tests. Furthermore, enabling the operator optimizations result in a consistent Mutation Score of around 67%. SFR can be a useful addition to complement the traditional BOR operator.

TOR - Transaction Origin Replacement Lastly, the TOR operator generated a large number of mutants, most of which survived testing. Since TOR swaps the `tx.origin` property with `msg.sender`, the fault is only noticeable under certain circumstances. In fact, the two properties only carry a different value within a chain of transactions.

> **Answering RQ1:** *Do the SuMo-specific and Solidity-specific operators contribute to the identification of weaknesses in Smart Contract test suites?*
>
> The novel and Solidity-specific operators contribute significantly to the total number of undetected mutants. These operators reveal gaps in existing test suites, particularly in areas like *event emission*, *exception handling* and *function modifiers*. Testers seem to often overlook Solidity-specific mechanisms in smart contracts, and can benefit from the additional support that mutation testing provides.

6.7.2 Operator Optimizations

In this section, we compare the Non-Optimized and Optimized mutation testing runs in terms of unique live mutants, subsumed mutants, and time costs, to answer RQ2: "*Can operator optimizations speed up the mutation testing process without compromising the adequacy assessment?*".

As shown in Table 6.16, both Non-Optimized and Optimized operators deliver similar perceptions of the test suite's quality, as reflected by their nearly identical mutation scores. The Non-Optimized run achieved a Mutation Score (MS_n) of 61.9% across 3,782 mutants, while the Optimized run yielded a similar Mutation Score (MS_o) of 61.5% from 2,528 mutants. This suggests that the Optimized set of operators maintains a comparable evaluation of test suite adequacy, even with fewer mutants.

The slight edge of the Non-Optimized run in generating more live mutants (L_n) might provide additional value in identifying potential weaknesses in the test suite. Therefore, while the Optimized set proves effective for general testing, the Non-Optimized operators may still offer a more thorough assessment, particularly in critical contexts where higher reliability is necessary.

Table 6.20: Potential Information Loss of Optimized Operators

DApp	$\lvert\Delta T\rvert$	$\lvert\Delta L\rvert$
`Alice`	245	89
`Blackjack`	323	222
`Coinosis`	72	8
`EtherCrowdfunding`	286	69
`Safe Contracts`	328	77
Tot.	1254	465

Information Loss Table 6.20 shows the difference in the number of mutants tested between the Non-Optimized and Optimized runs ($\Delta T = |T_n - T_o|$), and the unique live mutants generated exclusively by the Non-Optimized operators ($\Delta L = |L_n - L_o|$). The Non-Optimized set generated 1,254 more mutants than the Optimized set, of which 456 survived testing. This indicates that the Non-Optimized operators can uncover additional information about the test suite's quality. However, this additional information comes at a cost. The Non-Optimized set generates approximately 1.5 times more mutants than the Optimized set, leading to longer test execution times. While the Non-Optimized operators may capture more diverse fault-revealing mutants, further investigation is needed to assess whether T_n contains equivalencies that the TCE was not able to detect and if improvements to the test suites still bring similar scores for the two runs.

Subsumed Mutants To assess the efficiency of the optimizations, we analyzed the subsumption relationships among the tested mutants, as described in Section 6.6. We recall that, following the definition proposed by Kurtz et al. [94], one mutant $m1$ is subsumed by another mutant $m2$ if at least one test kills $m2$ and every test that kills $m2$ also kills $m1$. Reducing the number of subsumed mutants is a key objective of operator optimizations. Because there are no restrictions on the test set, we must approximate the subsumption relationships based on mutant behavior against the specific test suite of each project. Thus, we built the **mutant subsumption graph** containing all the subsumption relationships that always held for each DApp, and we used it to compute the subsumed mutants.

Table 6.21: Subsumed Mutants generated by SuMo

MutOp	Alice		Black.		Coin.		Ether.		Safe.	
	Su_n	Su_o	Su_n	Su_o	Su_n	Su_o	Su_n	Su_o	Su_n	Su_o
AOR	-	-	-	-	3	0	-	-	-	-
AVR	-	-	-	-	-	-	1	1	2	2
BOR	67	5	18	0	15	2	102	16	180	9
EHC	12	12	-	-	-	-	-	-	24	24
FVR	35	35	4	4	2	2	7	7	33	33
GVR	6	2	1	0	15	2	9	3	14	7
ILR	1	1	-	-	-	-	-	-	2	2
LSC	-	-	-	-	-	-	5	5	-	-
SFR	-	-	-	-	-	-	18	0	-	-
UORD	1	1	-	-	-	-	-	-	1	1
VVR	-	-	-	-	4	4	8	8	1	1
Total	122	56	23	4	39	10	150	40	257	79

Legend:
Su_o: Subsumed mutants generated by Non-Optimized operators,
Su_n: Subsumed mutants generated by Optimized operators

Table 6.21 presents the number of subsumed mutants (Su_n) generated by the Non-Optimized operators compared to those generated by the Optimized operators (Su_o).

The results show a significant reduction in the number of subsumed mutants when using Optimized operators. The BOR, GVR, and SFR operators seem to benefit the most from the minimal set of mutation rules. For instance, in the Safe-Contracts project, the Non-Optimized BOR operator produced 180 subsumed mutants, while its Optimized counterpart generated only 9. However, while the reduction in subsumed mutants is clear, further analysis is needed to determine how many potentially useful mutants are being excluded. It's also important to note that the number of subsumed mutants that can be discarded through optimizations can change depending on the implementation of the test suite. In general, it is not possible to select a subset of replacement rules that achieve perfect results for every project.

The analysis of the subsumption relationships also helped us to identify further improvement opportunities. As shown in Table 6.21, the EHC, FVR, ILR, LSC, UORD and VVR operators, which currently do not foresee an Optimized version, generated several subsumed mutants across projects. In Table 6.22 we show the specific couples of subsuming and subsumed replacement rules, where the mutants generated by the latter rule were always subsumed by the mutants generated by the former. The Table only shows those rules that hold for at least 3 out of 5 projects. These consistent subsumption relationships suggest potential for further optimization in operators like FVR and VVR, which do not currently have an Optimized version but often produce redundant mutants.

Table 6.22: Opportunities for operator optimizations

MutOp	**Subsuming Rule**	**Subsumed Rule**	**Projects**
FVR	public → internal	public → private	4/5
FVR	external → internal	external → private	3/5
VVR	public → internal	public → private	3/5

Time Costs Table 6.23 compares the time costs associated with generating and executing mutants in both the Non-Optimized and Optimized runs. The time to generate mutants is in the order of milliseconds and relatively unaffected by the optimizations. However, the time required for testing the mutants is significantly reduced by using Optimized operators. For instance, the Non-Optimized run for the Alice project required approximately 12 hours, whereas the Optimized version reduced this time by over 3 hours. On average, the Optimized operators decreased the overall test execution time by 30%. This demonstrates that operator optimizations can lead to considerable time savings, which is particularly beneficial for complex projects.

Overall, the non Optimized run could represent the best choice when DApp reliability is essential and the execution time is not a major concern. Indeed, the comprehensive set of rules of the Non-Optimized operators reduces the risk of skipping useful mutations but increases the number of subsumption relationships among mutants. The Optimized version instead, could be a good choice when the testing procedure should be faster, as may be the case in an incremental development context where the assessment could be run many times in a day.

Table 6.23: Time costs of mutation testing

DApp	Mutants Gen. Time (s)			Mutants Test Time (m)		
	Non-Opt	Opt	Δ	Non-Opt	Opt	Δ
`Alice`	1,13	1,08	*0,05*	720	537	*183*
`Blackjack`	0,99	0,95	*0,04*	205	162	*43*
`Coinosis`	0,48	0,42	*0,06*	86	55	*31*
`EtherCrowd.`	0,85	0,81	*0,04*	384	256	*128*
`Safe Contracts`	1,14	1,09	*0,05*	577	400	*177*

Answering RQ2: *Can operator optimizations speed up the mutation testing process without compromising the adequacy assessment?*

The use of Optimized operators resulted in similar mutation scores compared to Non-Optimized operators, while significantly reducing the generation of subsumed mutants and lowering the time cost by 30% on average. However, the Non-Optimized operators generate additional unique live mutants, which could be valuable in scenarios where code reliability is critical.

6.7.3 Improving a Test Suite using SuMo

In this section, we address RQ3: "*Can the live mutants help to improve the fault-detection capabilities of a test suite?*". To this end, we ran mutation testing on the Bionic Event project, and we manually improved its test suite based on insights gained from live mutants. To reduce the manual effort involved in analyzing subsumed mutants, we enabled operator optimizations during the experiment. After developing new or enhanced test cases, we re-ran mutation testing to evaluate the updated Mutation Score of the project.

Table 6.24: Mutation Testing Results for the Bionic Event DApp Before and After Enhancing its Test Suite

Test Suite	Mutation Operators	M	S	D	T	$MS(\%)$
Original	All	179	17	34	128	57
	Solidity	138	17	24	97	52,6
	SuMo	4	1	2	1	100
Enhanced	All	162	17	32	113	87
	Solidity	126	17	22	87	85,4
	SuMo	4	1	2	1	100

Legend:
M: total mutants, S: stillborn mutants,
D: equivalent and redundant mutants discarded by the TCE,
T: tested mutants, MS: mutation score

Table 6.24 summarizes the results of mutation testing before and after enhancing the test suite. The second round of testing yielded a significantly higher Mutation Score, demonstrating that the revised test suite had improved fault-detection capabilities. This improvement was driven by a manual analysis of the surviving (live) mutants from the first mutation testing run, which allowed us to create new tests and refine existing ones.

Several mutation operators were particularly useful in uncovering gaps in the original test suite. For instance, the *Binary Operator Replacement* (BOR) operator exposed issues related to boundary value testing, revealing that certain edge cases were not adequately covered. A BOR mutant also surfaced an unused function modifier, highlighting an opportunity for refactoring the smart contract itself.

```
1  function collectPayment() onlyOwner public {
2     selfdestruct(msg.sender);
3   ⊙ emit PaymentCollected(address(this), msg.sender,
       ↪ address(this).balance);
4   ★ //emit PaymentCollected(address(this), msg.sender,
       ↪ address(this).balance);
5  }
```

Listing 6.14: EED Mutant from the Bionic Event DApp

```
1  it("Should collect the payment for an event", async () => {
2
3    await firstEvent.purchaseTicket(1, { from: accounts[1], value:
       ↪ 3000000 });
4    let tx = await firstEvent.collectPayment();
5
6    truffleAssert.eventEmitted(tx, "PaymentCollected", (ev) => {
7       assert.equal(ev._event, firstEvent.address);
8       assert.equal(ev._organizer, accounts[0]);
9       assert.equal(ev._balance, 3000000);
10   });
11 });
```

Listing 6.15: Killer Test Case for the EED Mutant

Another particularly impactful operator was *Event Emission Deletion* (EED), which generated four surviving mutants. These mutants tell us that the original test suite rarely validated the emission of events. Events in Solidity serve as a communication channel between the blockchain and the outside world (e.g., off-chain systems such as front-end applications and monitoring tools). Testing for proper event emissions is essential for ensuring that smart contracts behave correctly in the broader application ecosystem and that users receive accurate feedback about the actions they've taken.

Among the EED mutants, one pointed to an actual bug in the contract's logic. This mutant targeted the `collectPayment()` function (shown in Listing 6.14), which attempts to emit an event immediately after executing a selfdestruct instruction. However,

once a contract is destroyed via selfdestruct, subsequent instructions (including event emissions) are never executed. As a result, the `PaymentCollected` event would never be emitted in practice, even though it appears in the source code.

To fix this issue, we moved the event emission statement to occur before the selfdestruct call, ensuring it is successfully executed. In addition, we developed a new test case specifically designed to kill the mutant and verify the correct behavior. This test is shown in Listing 6.15. It begins by initiating a transaction in which `accounts[1]` (representing a generic buyer) purchases a ticket from the `firstEvent` contract (line 3). After the purchase, the test simulates the contract owner's action by calling `collectPayment()` (line 4). Once the transaction is complete, the test checks whether the `PaymentCollected` event was emitted with the correct parameters (line 6). This is done using the `eventEmitted` assertion function provided by the `truffle-assertions` library.

```
1 function purchaseTicket (uint quantity ) public payable {
2   ⊙  require(quantity <= MAX_PURCHASE);
3   ★ //require(quantity <= MAX_PURCHASE);
4     require(available  >= quantity);
5     require(msg.value >= ticketPrice.mul(quantity));
6    ...
7 }
```

Listing 6.16: EHC Mutant from the Bionic Event DApp

```
1 it("should revert when buying more than 5 tickets at once", async() => {
2     await catchRevert(
3       firstEvent.purchaseTicket(6, {
4         from: accounts[1],
5          value: "3000000"
6       })
7     );
8 });
```

Listing 6.17: Faulty Test Case from the Bionic Event DApp Test Suite

Some mutants generated by the *Exception Handling Change* (EHC) operator survived due to flaws in the original test cases. Listing 6.16 shows one of the live EHC mutants, while Listing 6.17 presents the corresponding test case from the Bionic Event test suite that was intended to kill it. This test attempts to purchase more tickets than the maximum allowed for a single transaction. Under normal circumstances, this should trigger a revert at the first require statement in the `purchaseTicket` function (see Listing 6.16). However, the EHC operator removed this require statement during mutation, meaning the function should no longer revert, and the test should fail. The fact that the mutant survived indicates that the test still passed, even though the check it was supposed to verify had been removed. Upon investigation, we found that the test was not failing because

of the missing condition, but due to an unrelated issue: it did not send enough funds to purchase six tickets. As a result, the transaction reverted due to a different require check related to payment (line 5), not because of the ticket quantity restriction. This caused the test to pass incorrectly, masking the absence of the intended exception check.

To correct this, we updated the test by increasing the amount of wei sent with the transaction (line 5 in Listing 6.17), ensuring that the only failing condition was the ticket limit. With this fix, the test correctly failed when the require was removed and passed when it was present, successfully killing the mutant. Additionally, we designed new test cases to cover other edge scenarios, such as ensuring that a user cannot purchase more tickets than are currently available for an event. These enhancements improved the quality and reliability of the test suite, ensuring that important exception-handling logic is explicitly and accurately tested.

```
1 ⊙ function isTicketValid(address _owner,
2 uint _tokenId) onlyOwner external {
3 ⋆ function isTicketValid(address _owner,
4 uint _tokenId) ~~onlyOwner~~ external {
5     if(ownerOf(_tokenId) == _owner) {
6         _burn(_tokenId);
7     }
8 }
```

Listing 6.18: MOD Mutant from the Bionic Event DApp

```
1  it("should revert when a buyer validates own ticket", async() => {
2
3     await firstEvent.purchaseTicket(1, { from: accounts[1], value:
        ↪ "1000000" });
4
5     var tickets = await firstEvent.getOwnersTicket(accounts[1]);
6
7     await catchRevert(
8      firstEvent.isTicketValid(
9        accounts[1],
10       tickets[0],
11       {from: accounts[1]})
12   );
13 });
```

Listing 6.19: Killer Test Case for the MOD Mutant

The *Modifier Deletion (MOD)* operator also contributed to the test suite's improvement. The live mutant shown in Listing 6.18 revealed that the unhappy path of the `onlyOwner` modifier was not covered by the original tests. Therefore, we designed a new test (see Listing 6.19) to ensure that a buyer cannot validate their own ticket. This test issues a transaction on line 3 to purchase a ticket from the `firstEvent` contract on

behalf of `accounts[1]` (a standard, non-owner account). The second transaction on line 5 retrieves the array of tickets owned by `accounts[1]`, which contains the previously purchased ticket. Finally, on line 7 the test invokes the `isTicketValid` function from `accounts[1]`, passing the purchased ticked as a parameter. Since only the owner of the contract is allowed to perform this operation, the test awaits for a transaction revert. As a result, the MOD mutant in Listing 6.18 is effectively killed by this test.

> **Answering RQ3:** *Can the live mutants help to improve the fault-detection capabilities of a test suite?*
>
> Using SuMo on a real-world project allowed us to improve its Mutation Score by ≈ 53%. The live mutants helped us to both identify faulty and missing test cases, and to uncover a bug in the smart contract code.

6.8 Threats to Validity

In this section we present the threats to the validity of our study, distinguishing four different aspects as defined by Runeson and Höst [147]: construct validity, internal validity, external validity, and reliability. Here we address such threats and explain the measures we employed to reduce the risk of bias.

Threats to Construct Validity The assumptions made during the definition of the experiment that may influence the validity of the outcomes are referred to as construct threats. A threat to the construct of our experiment concerns the nature of the selected case studies. To measure the effectiveness of SuMo we considered the Mutation Score achieved by the selected applications. However, the quality of the test suites shipped with the projects affects the results of our experiment; a low-quality test suite will surely let many mutants survive. It is difficult to quantify how much effort the developers put into building solid tests. To validate SuMo, we focused on applications released with high-coverage test suites, and real-world programs that are intended to be highly secure. For instance, Safe-Contracts features large numbers of economic transactions and an up-to-date, high-quality test suite.

Threats to Internal Validity The threats to internal validity are those factors that might affect the validity of the results. One of such threats is the inclusion of undetected equivalent and redundant mutants in the calculation of the Mutation Score. During the experiments, we employed the TCE to remove such mutants and obtain a more reliable adequacy value for the chosen projects. However, no automated technique can detect all equivalencies. This is a well-known problem of the approach that can be mitigated by experienced testers, but due to the huge number of mutants generated during the experiments, we were not able to manually analyze and discard such mutants.

Threats to External Validity The threats to external validity concern the generalizability of our findings. To validate SuMo we run mutation testing on a small set of open-source Solidity applications. Thus, our findings might not apply to other projects

that offer different mutation opportunities. To mitigate this risk, we tried our best to select heterogeneous applications belonging to different domains, and with test suites of different complexities. Although we choose 5 distributed applications, some of which (e.g., Alice and Safe-Contracts) feature a relatively big number of Smart Contracts and test cases, we recognize that they are insufficient to establish universal validity for the given results. On the same note, in order to practically assess the usefulness of the mutation operators, we used SuMo for improving the test suite of a single project (i.e., Bionic Event Dapp). Improving a test suite is a time-consuming activity that also requires extensive knowledge of the application under test. Analyzing the mutants helped us to improve the test suite and to spot a not previously reported bug on the project. Nevertheless, to derive a definitive answer SuMo must be adopted to improve more complex test suites, and on a larger set of projects where it is possible to apply all the mutation operators.

Threats to Reliability This category describes the extent to which the data is dependent on the conducted research, and whether the results can be reliably reproduced. The main threat to the reliability of our findings concerns the presence of flaky tests in the test suites shipped with the selected projects. These are tests that can exhibit pass and fail outcomes on different re-runs, although neither the code under test nor the test have changed. As shown by Shi et al. [151], the Mutation Score achieved by a project might not be the same on two identical re-runs if such tests are present. For instance, a mutation m might be killed by a test t during a run *r1*. If the same mutant m is not covered by the same test t during the next run *r2*, it will be marked as live. To mitigate the risk of unreliable Mutation Scores, we re-run each test suite on the original project several times to weed out applications with possible non-deterministic tests. Even so, we cannot guarantee the absence of such tests in the selected projects, as some flaky test behavior might be difficult to observe after a few re-runs, and might also be dependent on the injected mutation.

6.9 Related Work

Existing studies relevant to our proposal focus on two key areas, mutation testing, and specific mutation testing approaches for the Solidity language.

Mutation Testing Initially proposed by Richard Lipton [33], mutation testing ensures the adequacy of test cases in detecting faults by introducing artificial defects (i.e., *mutations*) into the code under test. Mutants have demonstrated their effectiveness as test goals, leading to increased adoption in industrial settings [136, 26] and across many programming languages (e.g., PIT [140] for Java, MutPy and Cosmic Ray [52] for Python). The literature on mutation testing has investigated and contributed many approaches for improving the effectiveness and usability of this technique. Jia et al's [87] study provided a comprehensive overview of the available work on cost reduction, while Kintis et al. [91] investigated techniques for reducing mutation testing expenses while maintaining its effectiveness (e.g., higher-order and weak mutation testing). Papadakis et al. [134] presented a survey of recent advances in mutation testing, exploring the main challenges to be tackled for the future development of the technique. Additionally, Papadakis et al.

introduced the Trivial Compiler Equivalence [133], which is largely used in modern tools for detecting equivalent and redundant mutants.

Mutation Testing for Ethereum Smart Contracts In the literature, it is possible to find some alternative proposals that implement a mutation testing approach for Ethereum smart contracts. Wu et al. propose *MuSC* [192, 100], the first mutation testing framework for the assessment of Ethereum smart contract test suites. This work proposes nine JavaScript-oriented operators and fifteen Solidity-specific mutation operators that can inject faults in the smart contract code. The proposed set of operators is moderately compact, which allows limiting the total number of generated mutants. However, it lacks several operators that can introduce severe faults or vulnerabilities into the contract, such as the ones that target function modifiers. While the mutation operators used by MuSC were designed starting from the Solidity documentation, the approach proposed by Andesta et al. [3] is based on the study of known bugs in Solidity smart contracts. This set of Solidity operators was included in *Universal Mutator* [72], a popular regexp-based mutation testing tool that can easily be adapted to different programming languages. The core idea of Andesta et al.'s work is that designing operators capable of recreating real-world bugs can be an effective way of selecting good test cases. This led to the definition of 10 classes of operators that cover many types of smart contract vulnerabilities. The results show that the proposed set leads to the generation of a big amount of invalid and redundant mutants, while the rate of generation of equivalent mutants was not mentioned. Chapman [40] proposes *Deviant*, an open-source mutation testing tool for Solidity. In this case, a broader set of 62 Solidity-specific mutation operators was designed according to the Solidity fault model. The *ContractMut* tool proposed by Hartel & Schumi [75] implements general mutation operators derived from the standard Mothra set, and four Solidity-specific mutation operators. In order to define a more manageable set of operators and limit the number of generated mutants, the authors apply *selective mutation* techniques. In particular, the mutation operators were designed to only target the most severe vulnerabilities, with a focus on mistakes concerning access control.

Smart Contracts flaws and vulnerabilities Several works analyze or classify bugs and vulnerabilities that can be found in Ethereum smart contracts. Atzei et al. [8] provide a comprehensive taxonomy of common programming flaws which may lead to actual vulnerabilities. The authors distinguish the vulnerabilities according to the level where they are introduced (Solidity, EVM bytecode, or blockchain), and they provide examples of real exploit patterns. Groce et al. [71] provide a summary of flaws detected in paid security audits performed at a leading company in blockchain security. Such analysis helps to understand the likelihood and severity of real smart contract flaws, many of which can be currently reproduced by SuMo. Indeed, the proposed mutation operators can simulate bugs and/or vulnerabilities of the following classes: Access Control, Numerics, Undefined Behavior, Authentication, Reentrancy, Error Reporting, Timing, Coding-Bug, Auditing and Logging, Missing Logic, and Cryptography. For instance, the mutation operators that target function modifiers can help to simulate those cases where access control is either too permissive or too restrictive. This is extremely relevant as access control issues seem to be amongst the highest severity bugs in Smart Contract development [71].

6.10 Directions for Future Research

Future research directions include evaluating the effectiveness of mutation operators that were not applied in our experiments, as well as implementing optimized versions of those that consistently produced subsumed mutants across multiple projects. Expanding the evaluation of SuMo to a broader range of smart contracts could also help identify common subsumption relationships among the implemented mutation rules. This, in turn, would inform further improvements to the operator optimization strategies. Understanding which subsumption rules tend to hold generally can reduce the number of redundant mutants without significantly increasing the risk of omitting valuable mutation scenarios.

Furthermore, to enhance the practicality of SuMo in real-world development workflows, it is important to add support for incremental mutation testing. As observed in the case of the Alice project, generating and evaluating a comprehensive set of mutants can be time and resource-intensive, leading to long delays before actionable feedback is available. While a full mutation testing campaign may be acceptable when performed occasionally, repeating the entire process frequently during development is often impractical.

To make mutation testing more scalable, especially for large and evolving codebases, future work should focus on integrating a regression mutation testing capability (as proposed by Zhang et al. [199]) into SuMo. This enhancement would enable SuMo to: (1) generate mutants only for the portions of the code that have changed since the last version, and (2) selectively run test cases that exercise those modified sections. Narrowing the scope of analysis to the most relevant code and tests would significantly reduce both execution time and computational overhead, making SuMo more suitable for continuous integration environments and more viable for adoption in real-world smart contract development.

CHAPTER 7

RESUMO: REGRESSION MUTATION TESTING FOR SOLIDITY SMART CONTRACTS

Mutation testing is a powerful test adequacy assessment technique that can guarantee the deployment of thoroughly tested and more reliable smart contract code. However, re-evaluating the test suite after the introduction of tests or code changes can be unfeasible due to the high costs of the assessment. This Chapter introduces ReSuMo, the first regression mutation testing approach and tool specifically designed for Solidity Smart Contracts. ReSuMo applies a static, file-level selection technique to identify those smart contracts and test files that can be affected by recent changes. Then, it incrementally updates mutation testing results using the mutants from the previous revision. In this way, ReSuMo can help to speed up test adequacy assessment while still providing a reliable evaluation of the test suite's quality.

This chapter is based on material from the following publications: [15, 24]

* * *

7.1 Introduction

Smart contracts often handle business-critical logic and have unique lifecycle constraints, making rigorous testing essential. During software development, test suites should be continuously updated to reflect changes in the smart contracts. At the same time, developers should assess whether their test suites remain adequate as the codebase evolves. In Chapter 6 we introduced SuMo, a mutation testing framework for Solidity smart contracts. SuMo evaluates and improves test adequacy by injecting various types of faults into the contract code. Mutation testing provides a much stricter measure of test quality compared to simple code coverage [64], making it a powerful resource for contract developers.

Despite its effectiveness, mutation testing remains an underused technique among Ethereum developers due to its high computational cost and limited integration into existing development workflows. In practice, most mutation testing approaches [3, 40, 100, 75] are designed as standalone processes, executed only when a test suite reaches maturity or just before deployment. This traditional perspective creates two major challenges. First, running continuous mutation testing throughout development is impractical and time-consuming. As shown in our previous study [22], a full mutation testing run can require many hours to be completed. As project complexity increases, the cost of mutation testing scales accordingly, making *frequent re-evaluations infeasible*. As a result, developers are forced to postpone mutation testing until late in the development cycle, where addressing all surviving mutants at once can become overwhelming. For similar reasons, mutation testing is impractical to apply efficiently to new releases of upgradeble smart contracts (e.g., those using the proxy pattern). Ideally, each new release should undergo extensive validation, but re-running a full mutation testing campaign from scratch can be excessively time-consuming, delaying important security updates and bug fixes. Under these conditions, developers might decide to forgo test suite re-evaluation.

In this Chapter we present ReSuMo (a mutation of REgression SOlidity MUtator). ReSuMo is an evolution of the SuMo framework that implements the first *regression mutation testing* approach for Ethereum smart contracts. The core idea behind Regression Mutation Testing (RMT) is that not all mutants and test executions need to be recomputed when a software project is modified. Instead, it is possible to selectively update mutation testing results for a new project revision by reusing relevant test-mutant execution outcomes from a prior revision [200]. Since RMT only re-evaluates a subset of tests, it can significantly lower execution times and make continuous adequacy evaluation more feasible. RMT is inherently flexible and can be applied at different stages of the development lifecycle: after a single commit, at designated checkpoints, and between contract releases. However, the intersection of regression testing and mutation testing has only been explored in traditional software engineering contexts (e.g., ReMT [200] for Java programs). Applying RMT in blockchain development introduces strict constraints. While all RMT techniques aim to balance efficiency and safety [42], the latter is crucial when dealing with smart contracts. Safe RMT frameworks ensure that selective test execution does not introduce precision issues, and that the final mutation score is reliable.

We therefore conduct a study to investigate the effectiveness of ReSuMo in reducing mutation testing costs without compromising the precision of its results. To this end, we apply ReSuMo to successive commits of real-world Ethereum projects from GitHub. First, we analyze the *efficiency* of ReSuMo by measuring its time savings compared to running full mutation testing. The results show that ReSuMo can support frequent test adequacy assessment, with an average time savings of almost 60% on new project revision. Second, we evaluate whether the selective execution strategy of ReSuMo is *safe*. We find that the mutation score obtained via regression mutation testing is still precise despite avoiding full test re-evaluation.

The rest of this chapter is organized as follows: Section 7.2 introduces the research questions that we aim to answer in this thesis. Section 7.3 describes the proposed regression mutation testing approach, whereas Section 7.4 describes the key differences in the design and implementation of ReSuMo with respect to SuMo. Sections 7.5 and 7.6 describe the experiment methodology and present the discussion of results, while Section

7.7 discusses the threats to the validity of this study. Lastly, Section 7.8 reports related work, and Section 7.9 identifies areas for further research.

7.2 Research Questions

This thesis introduces ReSuMo, a novel regression mutation testing framework for Solidity smart contracts. ReSuMo is designed to optimize mutation testing for evolving smart contracts by selectively reusing previous test-mutant execution results. Its goal is to make mutation testing more practical for real-world development by reducing execution time while maintaining the precision of test adequacy assessment. In the following, we introduce the two research questions that guide this research.

RQ1: Can ReSuMo reduce the time cost of full mutation testing in evolving smart contract projects? RQ1 aims to evaluate whether ReSuMo can reasonably reduce the time cost of continuous test adequacy assessment in evolving smart contract projects. Full mutation testing can be prohibitively expensive when executed on every new project revision, particularly for complex smart contracts with extensive test suites. Developers may therefore be discouraged from performing frequent assessments, potentially compromising code quality. To address RQ1, we conduct an empirical evaluation by applying ReSuMo to consecutive revisions of real-world Solidity projects. The same assessment is then performed using SuMo, which does not implement any regression mutation testing mechanism. If ReSuMo substantially reduces the computational expense of full mutation testing, it could encourage developers to regularly refine their test suites in parallel with code evolution, promoting the deployment of more reliable smart contracts.

RQ2: Does ReSuMo produce precise mutation testing results in evolving smart contract projects? RQ2 aims to establish whether ReSuMo can provide precise adequacy feedback to developers. Ideally, a test suite should yield the *same mutation score* regardless of whether regression mutation testing or full mutation testing is used. Any discrepancies in mutation scores could mislead developers about the adequacy of their test suite, providing distorted feedback that may complicate test improvement efforts. For instance, if a test suite successfully detects and kills a mutant, but ReSuMo incorrectly classifies it as live, developers may falsely believe that their test suite has a gap. This could lead them to spend unnecessary effort writing additional tests that are, in reality, redundant. Instead, if a mutant that should remain live is incorrectly classified as killed, developers may assume their test suite is more effective than it actually is. Over multiple software revisions, accumulated errors could give a misleading impression of test quality to developers. The precision of the results inherently depends on the safety of the proposed regression mutation testing approach. To answer RQ2, we analyze the mutation scores computed in the same experiment described for RQ1. If the mutation scores obtained via ReSuMo closely match those obtained via full mutation testing, we can conclude that ReSuMo provides reliable adequacy feedback.

7.3 The ReSuMo Framework

The SuMo framework, introduced in Chapter 6, implements full mutation testing for Solidity smart contracts. This means that, following any codebase modification, it necessitates the re-evaluation of all tests on all mutants from scratch. In this Section, we present ReSuMo, an evolution of the SuMo framework that implements a *Regression Mutation Testing* (RMT) approach. Regression mutation testing was first introduced by Zhang et al. [200] as a technique to incrementally update mutation testing results across software revisions. Consider the scenario where a codebase is modified and evolves from revision R to R′. The key idea behind RMT is that some *mutant-test* execution pairs from R can be safely reused in R′, reducing redundant executions. In particular, RMT is based on two key concepts:

1. *Regression Test Selection*: As discussed in Section 3.7, RTS has been widely used to speed up regression testing activities. RTS identifies a subset of tests that must be *re-executed* in response to modifications in the program under test. There exist many different RTS strategies, but most state-of-the-art tools focus on selecting those *modification-traversing tests* that execute code regions impacted by changes [67, 97]. Similarly, RTS can be used to speed-up mutation testing in evolving systems [42], as it allows identifying a subset of tests to be re-evaluated on mutants of a new program revision R′.

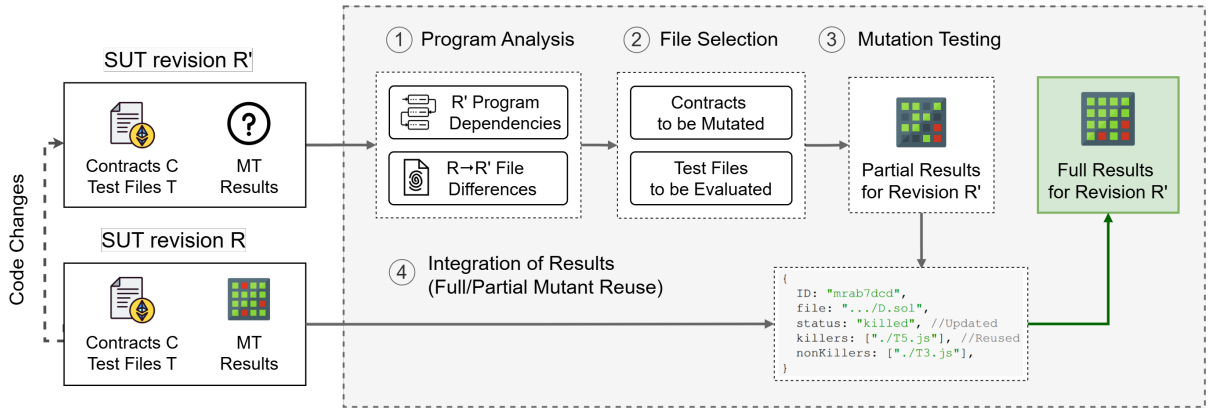
2. *Incremental Update of Mutation Testing Results:* When mutation testing runs on a subset of all possible tests and mutants, the resulting mutation score represents a partial measure of test adequacy. Consider the following scenario: In project revision R, a mutant m was tested against the entire test suite T, and a test case t successfully killed m. Then, RTS is applied to determine a subset of tests in T to be re-executed for the new revision R′. If t is not selected for re-evaluation, m could be incorrectly classified as *live* in the new revision. As a result, not only the mutation score for R′ will be imprecise, but testers will also waste time inspecting a misclassified live mutant. Thus, RMT approaches identify reusable mutant-test outcomes from the prior revision R that can must be carried over to R′.

In the following, we detail each aspect of the proposed RMT approach. Section 7.3.1 begins with a high-level description of ReSuMo and of its application context, while Section 7.3.2 provides the rationale behind its static, file-level approach to regression mutation testing. Section 7.3.3 then explains how ReSuMo computes program changes and dependencies, while Sections 7.3.4 shows how it identifies contracts and regression tests for the evaluation. Finally, Sections 7.3.5 and 7.3.6 discuss how ReSuMo performs mutation testing and how it incrementally updates the results.

7.3.1 Regression Mutation Testing Approach

Consider a project composed of a set of smart contracts C, and a set of test files T, as shown in Figure 7.1. When a developer commits changes, the project transitions from revision R to revision R′. These changes may involve modifications to smart contracts in C, updates to test cases in T, or even the addition or removal of files. The mutation

Figure 7.1: Regression Mutation Testing Approach implemented by ReSuMo



testing results for revision R are available, but due to the introduced changes, the test suite of revision R′ must be re-evaluated. The objective of ReSuMo is introducing a RMT approach that lowers the time costs of incremental adequacy evaluation without compromising safety. Here, *safety* refers to the guarantee that all mutant-test pairs that could yield different results due to code changes are re-executed on R′. On the other hand, *efficiency* measures how well the approach reduces execution time. Balancing safety and efficiency is a key problem in RMT [42], and it heavily depends on the chosen RTS strategy and the precise reuse of mutation testing results. Given the high-stakes nature of Solidity smart contracts, ReSuMo introduces a more conservative RMT approach that errs on the side of running more test-mutant pairs rather than risking the omission of critical ones. In this way, it mitigates the risk of imprecisions in the mutation testing results. In particular, ReSuMo applies a static, file-level regression mutation testing approach that comprises four key phases:

1. **Program Analysis:** The program analysis phase extracts necessary artifacts to determine which contracts and tests require re-evaluation. Like any RTS tool that must compute modification-traversing tests, ReSuMo identifies:

   - The **differences** between contracts and test files in revisions R and R′.
   - The **program dependencies** in revision R′.

2. **File Selection:** Using the extracted artifacts, ReSuMo then applies a selection strategy to identify:

   - A subset of **test files** in T that must be *re-evaluated*.
   - A subset of **smart contracts** in C that must be *mutated*.

   ReSuMo complements test file selection with smart contract selection to reduce the number of mutants to be generated and tested. Indeed, if a smart contract in R′ is unaffected by the latest changes, it will certainly generate the same mutants as in the previous revision R. Thus, if tests of unchanged smart contracts do not need to be re-evaluated, ReSuMo can directly reuse prior results for these files.

3. **Mutation Testing:** When running mutation testing on R′, ReSuMo limits mutant generation and test execution to only the selected files. Specifically, it only generates mutants for the selected smart contracts, and tests each mutant with the selected test files that target its contract. This process results in *partial mutation testing results* for R′.

4. **Integration of Mutation Testing Results:** The final step ensures that the mutation score remains precise despite the selective execution. ReSuMo merges the partial mutation testing results of R′ with the prior results from R, producing complete results for R′. This integration accounts for mutants whose test outcomes can be *fully* or *partially* reused, but this aspect will be discussed in Section 7.3.6 in more detail.

Before presenting how ReSuMo works in detail, it is important to clarify what *software revision* means. In regression mutation testing, the breadth of a software revision is not set in stone. Real projects evolve through a series of changes that can vary significantly in scope. While some changes consist of minor bug fixes or refactorings, others may introduce entirely new features. In practice, each such modification is captured as one or more commits to the project's codebase. Regression mutation testing is inherently flexible and can be applied at different stages of the development life-cycle: it can be invoked after a single commit, at designated checkpoints, or in conjunction with continuous integration workflows. Therefore, we define a *revision* as a new version of the project after the introduction of a *change*, where a *change* is a set of one or more *commits* grouped at the discretion of the developer. This definition allows teams to decide how granular each testing cycle should be.

7.3.2 Static and File-Level Regression Mutation Testing

Regression test selection techniques use revision differences and program dependencies to build a safe regression test suite for each new version of a program. As introduced in Section 3.7, in the context of regression testing, *safe* means that if a test can potentially reveal a fault introduced by recent changes, it is included in the regression suite. Extending this idea to regression mutation testing means selecting all the tests (and contracts) whose re-evaluation (and mutation) might result in a different mutation score. While efficiency is a key consideration in RMT, safety takes priority in high-stakes domains such as blockchain. Both these properties of RMT are affected by the chosen computation granularity and the specific techniques used for collecting program dependencies. In the following, we discuss why ReSuMo adopts static analysis and file-level granularity, considering both their advantages and practical feasibility.

File-Level Regression Mutation Testing

A key decision that affects any approach based on regression test selection is the granularity of computation at which it operates. Granularity refers to the size of the program units considered for dependency analysis and change detection. This choice dictates the selection of tests to be re-evaluated and, in the case of ReSuMo, the selection of contracts to be mutated. The following approaches to RTS exist:

Table 7.1: Comparison of RTS Techniques: Ekstazi, STARTS, and FaultTracer

RTS Tool	Technique	Granularity	Safety Considerations
Ekstazi	Dynamic	File-level	May miss dependencies due to execution path divergence, mitigated by file-level analysis.
STARTS	Static	File-level	Over-approximates file-level dependencies, ensuring safer test selection.
FaultTracer	Dynamic	Method-level	May miss dependencies due to execution path divergence, more severe at the method-level.

- *Basic block–level RTS* selects only the tests that exercise the exact statements or lines of code that have changed. If a single line in a method is modified, only the tests that reach that line will be included in the test suite.

- *Method-level RTS* treats entire functions as the atomic units. A small modification to a statement causes the entire function to be marked as changed. Thus, any test that invokes the function (whether or not it executes the altered statement) would be selected. Like for the basic block-level, this technique can produce minimized and efficient test suites, because it rarely picks more tests than necessary.

- *File-level RTS* treats entire files as the atomic units. This coarser granularity is more conservative, as even a minor modification to a file (e.g., a smart contract, a Java class file) triggers all related tests to be re-executed. Therefore, file-level RTS may select a larger regression test suite than method or basic block–level.

ReSuMo adopts a *file-level* approach to regression mutation testing. The reasons behind this choice mostly concern safety, but also efficiency and practical applicability.

Safety: File-level RTS reduces precision issues in mutation testing compared to method-level approaches. A comprehensive study by Chen et al. [42] examined the impact of popular Java RTS tools (FaultTracer [198], Ekstazi [67], and STARTS [97], as summarized in Table 7.1) in speeding up mutation testing. The authors investigated how different test dependency granularities affect RTS-based regression mutation testing. The study found that method-level RTS can fail to detect new interactions between test cases and changed code due to limitations in dependency tracking. A test may execute additional method-level entities that were not originally identified, leading to imprecision. Instead, file-level RTS does not suffer from this issue to the same extent, as it is harder to diverge a test execution to execute other changed file-level entities.

Efficiency: Despite its conservative selection strategy, file-level RTS remains efficient in practice. The same study by Chen et al. [42] demonstrated that both static and dynamic file-level RTS approaches can reduce execution time by approximately 95% compared to traditional mutation testing. Furthermore, the end-to-end time of regression mutation testing does not only include test execution, but also the overhead of dependency collection, change analysis, and any other operation. Method-level techniques typically require precise tracking of dependencies and changes, which can be costly. The computational overhead of file-level RTS is minimal, as it relies on simple

checksum comparisons, making it lightweight and scalable.

Applicability: File-level RTS is significantly easier to implement and maintain than finer-grained techniques. Ekstazi, for example, tracks dependencies dynamically by monitoring file-level accesses and computing checksums, avoiding expensive code instrumentation. Instead, the foundational ReMT [200] technique has never seen practical tool support due to its complex design and fine-grained analyses. Moreover, file-level RTS aligns naturally with the structure of smart contract test suites. Ethereum testing frameworks, such as *HardHat*, execute tests at the file level rather than at the method level. Smart contracts resemble classes in object-oriented programming, with test suites typically designed around individual contracts. Enforcing finer test selection would introduce unnecessary complexity and potential reliability issues. In fact, smart contracts maintain persistent state across transactions, increasing the risk of order-dependencies between test methods. File-level RTS ensures that all test methods are re-executed in the designed order and with the relative set-ups when a contract file is modified.

Static Regression Mutation Testing

In regression mutation testing, program dependencies are used to determine which test files must be selected for re-evaluation, and which contracts for mutation, whenever the codebase evolves. Program dependencies can be collected statically or dynamically:

- *Static RTS* gathers program dependencies by analyzing the source code of the system under test;
- *Dynamic RTS* collects program dependencies by monitoring test execution on a prior revision of the system under test.

ReSuMo adopts a *static* approach to dependency collection as it aligns well with its file-level granularity. Like for the computation granularity, the choice of static analysis influences RMT in terms of efficiency, safety and applicability.

Safety: Static RTS is inherently safer than dynamic RTS because it *over-approximates* dependencies, ensuring that all potentially affected test files are flagged for re-execution. Dynamic RTS collects dependencies by executing tests, and can miss certain dependencies due to runtime conditions.

Efficiency: Despite its conservative nature, static RTS has been shown to be competitive with, and in some cases superior to, dynamic RTS in terms of efficiency. The study evaluating STARTS (static) and Ekstazi (dynamic) demonstrated that both approaches achieve comparable reductions in test execution [42]. Moreover, dynamic RTS tools must execute all tests in the prior revision to track dependencies, adding runtime costs.

Applicability: Dynamic RTS techniques require monitoring test execution, which is more impractical in real-world development environments. Solidity's diverse testing

frameworks generate coverage reports in different formats, often providing only aggregated contract-level coverage without mapping test files to specific executed code. Building such mapping would require re-executing each test file independently, an approach that is computationally expensive. Static RTS is more scalable and lightweight, especially when working at the file-level. In fact, it would simply focus on import statements, making it easier to adapt to the many testing languages in Solidity's ecosystem.

7.3.3 Analysis of File Changes and Dependencies

The regression mutation testing workflow of ReSuMo on a new project revision R′ starts with the program analysis phase. As shown in Figure 7.1, this phase involves the computation of program dependencies and the computation of file changes with respect to the previous revision R. In the following, we describe each of these activities in more detail.

Computation of File Changes

ReSuMo operates at the file-level, so the differences between any two revisions R and R′ can be quickly computed with a simple checksum. When the mutation testing process on R′ begins, ReSuMo calculates and stores the hash of each contract and test file in the project. Listing 7.1 shows an example entry for a smart contract file. If the entry includes a *lastChecksum* field, it means that the file underwent mutation testing in the prior revision R, thus its value is compared to the current *checksum* to detect changes. If the file has no *lastChecksum* it is either new, or there are no prior mutation testing executions for the project, thus it is automatically marked as changed.

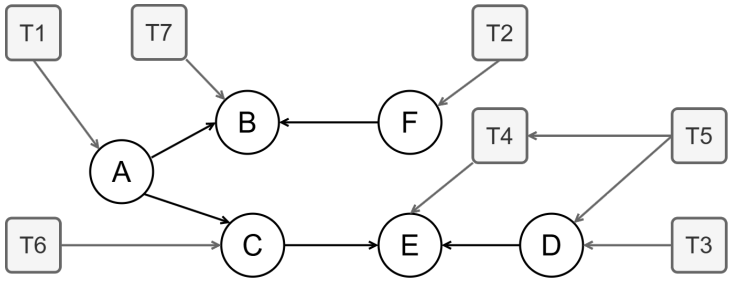
```
{
   filePath : ".../Contract.sol",
   checksum: "6aba8ee1fe7271a2240945ccae3f5219e7807696",
   lastChecksum: "86fc7a0fdbf508845c5305ced3eea4e27e2e6497"
}
```

Listing 7.1: Example of Checksum File saved by ReSuMo

Computation of Program Dependencies

ReSuMo collects program dependencies statically. To this end, it builds a dependency graph representing the system under test, where each file is stored as a node, and each dependency as an oriented arc connecting two nodes. To clarify how the proposed strategy works, we show in Figure 7.2 the dependency graph generated by ReSuMo for a simple Solidity project. First, ReSuMo adds one node to the dependency graph for each test and for each contract file. Here, the white circles represent Solidity smart contracts, and the gray squares represent the test files. Then, it parses and visits each file in search of *Import Directives*. These are statements that let a file import another file and can therefore be used to approximate the project dependencies.

Figure 7.2: Dependency Graph generated by ReSuMo for a simple Solidity project



For instance, let's consider the JavaScript test file `T5` shown in listing 7.2. The `artifact.require` import directive on line 2 lets `T5` use contract `D`, while the `require` statement on line 3 imports another test file `T4`. ReSuMo uses such statements to quickly establish direct dependencies among files.

```
1  /* Test File T5 */
2  var D = artifacts.require("D");
3  require ("./ T4");
4
5  contract ("D", function (accounts) {
6      it("should do something ...", async function () {
7          d = new D(some_address);
8          ...
9      })
10 })
```

Listing 7.2: A Test File T5 that imports a Contract D and a Test File T4

In Ethereum development, smart contracts are primarily written in Solidity, but tests can be written in many different languages depending on the used testing framework. ReSuMo is currently compatible with *Truffle* and *HardHat*, thus it implements static analysis for test written in JavaScript and Solidity. In the future, it could also be extended to support other languages. Once the dependency graph is built, ReSuMo can use it to quickly extract information about each node. In particular, the used dependency graph utility[1] exposes utility methods to extract:

1. *File Dependencies*: the files that the specified node depends on (transitively);

2. *File Dependants*: the files that depend on the specified node (transitively).

ReSuMo saves the extracted information to file in the form of a *dependency mapping*. An example of dependency mapping for the test file `T5` is shown in Listing 7.3. When the mutation testing process starts, ReSuMo uses the information stored in this mapping to determine which test files should be run on each mutant.

[1]Dependency graph: `https://www.npmjs.com/package/dependency-graph`

```
{
  file: ".../test/T5.js",
  testDependencies: [".../test/T4.js" ],
  contractDependencies: [".../contracts/D.sol"],
  testDependants: [],
  contractDependants: []
}
```

Listing 7.3: Dependency Mapping generated by ReSuMo for Test File T5

7.3.4 Selection of Smart Contracts and Test Files

Let's consider the scenario introduced in Section 7.3.1. We define the following:

- C the set of all smart contract files in revision R′;
- T the set of all test files in revision R′;
- $C' \subseteq C$ the sets of contract files that have changed between R and R′;
- $T' \subseteq T$ the sets of test files that have changed between R and R′;
- $dependencies(C')$ the set of contracts that are used by any contract in C';
- $dependents(C')$ the set of contracts that use any contract in C'.

One key objective of ReSuMo is determining a subset of contract files $C_{\text{mut}} \subseteq C$ that must be mutated again, and subset of test files $T_{\text{eval}} \subseteq T$ that must be re-evaluated. In the following, we describe the relative selection strategies.

Identifying Smart Contracts to be Mutated

Selecting a subset of smart contracts to be mutated can significantly reduce time and resource consumption. However, this selection must be done carefully to preserve safety guarantees. Since test files are evaluated and improved based on their ability to kill mutants, an imprecise selection of contracts for mutation can result in inaccurate mutation scores and ultimately weaker test suites. The central question is:

Which mutants generated from a new project revision must be (re)tested?

A mutant generated from a new project revision R′ must be generated and tested if its behavior may differ from that of the previous revision R. Several situations can lead to such differences and must be correctly identified by ReSuMo. Assuming the absence of flaky (i.e., *non-deterministic*) tests, we can generally make the following assumptions:

1. *Unchanged smart contracts* generate the same mutants;
2. *Unchanged test files* produce the same results on the same mutants, provided that no modified statements are executed.

The contract selection strategy implemented by ReSuMo takes as input the changed files $C' \cup T'$ along with a precomputed *dependency mapping*, and outputs a subset of smart contracts to be mutated $C_{\text{mut}} \subseteq C$. We identify three contract selection scenarios based on the nature of the changes introduced by a commit: (1) *Only Changed Contracts*, (2) *Only Changed Test Files*, (3) *Changed Contracts and Test Files*. This strategy is inspired by recent work on Commit-Aware Mutation Testing [104], which shows that mutating only the evolved code can overlook critical interactions with unmodified code. Consequently, ReSuMo also includes potentially affected but unchanged contracts in its mutation scope.

(1) Only Changed Contracts: $C' \neq \varnothing$, $T' = \varnothing$

Any contract $c \in C'$ can generate some new mutants with respect to the previous revision R. Therefore, C_{mut} will include:

- the set of changed contracts C';
- $\mathit{dependencies}(C') \cup \mathit{dependents}(C')$;

The latter might exhibit different behavior due to the evolution of the contracts in C' and should be retested accordingly.

(2) Only Changed Test Files: $C' = \varnothing$, $T' \neq \varnothing$

In this case, the contract files have not changed, so all contracts $c \in C$ will generate the same mutants as in R. However, the updated test files $t \in T'$ may behave differently against the same mutants. Therefore, C_{mut} will include any contract $c \in C$ that is directly or indirectly invoked by any $t \in T'$.

(3) Changed Contracts and Test Files: $C' \neq \varnothing$, $T' \neq \varnothing$

When both contracts and test files have changed, we combine the previous strategies. C_{mut} will therefore include:

- $C' \cup \mathit{dependencies}(C') \cup \mathit{dependencies}(C')$;
- any contract $c \in C$ that is used (directly or indirectly) by a test file $t \in T'$.

Identifying Tests to be Evaluated

When a project transitions to a new revision R′, the adequacy of its test suite may be affected by changes to smart contract and test files introduced since R. In the context of regression mutation testing, the primary goal is to *avoid redundant test executions* by identifying and skipping tests that were already assessed in R and that cannot possibly produce different outcomes on the mutants of R′. Instead of re-running such tests, their previous results (i.e., *pass* or *fail*) can be safely reused when computing the new mutation score. To achieve this, we must answer the following question:

Which tests can produce different outcomes on mutants of a new project revision?

We define the set of test files to be (re)evaluated in R′ as follows:

$$T_{\text{eval}} = T' \cup \{t \in T \mid t \text{ interacts with some } c \in C' \cup \mathit{dependents}(C')\}$$

That is, a test file must be re-evaluated if it has changed or if it interacts with any changed contract or its dependents. These two cases are detailed as follows:

(1) New or Modified Test Files: Any test file $t \in T'$ (any file added or modified since R) may exhibit new behavior and must therefore be re-executed. Newly introduced test files have no prior mutation testing results and must be fully evaluated.

(2) Test Files That Exercise Changed Contracts or Their Dependents Even if a test file $t \notin T'$ is itself unchanged, it must still be re-evaluated if it exercises:

- a changed contract $c \in C'$
- any contract $c \in \mathit{dependents}(C')$ (i.e., contracts that use a changed contract).

These interactions may affect the behavior of the test in the new revision. In contrast, test files that only interact with *dependencies* of changed contracts (without exercising the changed contracts themselves or their dependents) can be safely excluded from re-evaluation.

7.3.5 Mutation Testing

Once the relevant smart contracts C_{mut} and test files T_{eval} are selected, ReSuMo initiates the mutation testing process. The overall methodology follows the approach used in SuMo, as described in Chapter 6, but with one fundamental difference. While SuMo determines the status of a mutant based on the *aggregated outcome* of all executed test files, ReSuMo must track the outcome of each individual test file against each mutant. Since ReSuMo reuses mutant-test file pairs across revisions, it must distinguish which test files are responsible for detecting a given mutation. SuMo already maintains structured mutation data that is updated at run-time.

An example of a mutation object is shown in Listing 7.4. Besides its *ID*, a mutation is characterized by the *file* from which it was generated and by its *status* (e.g., *killed* or *live*). ReSuMo extends each mutation object to include the list of its *killers* (i.e., the test files for which at least one method failed on the mutant) and *nonKillers* (i.e., the test files that passed on the mutant without triggering a failure). This list of mutation objects is functionally equivalent to a *mutation matrix* [200], a structured representation of all mutant-test execution pairs.

7.3.6 Integration of Results

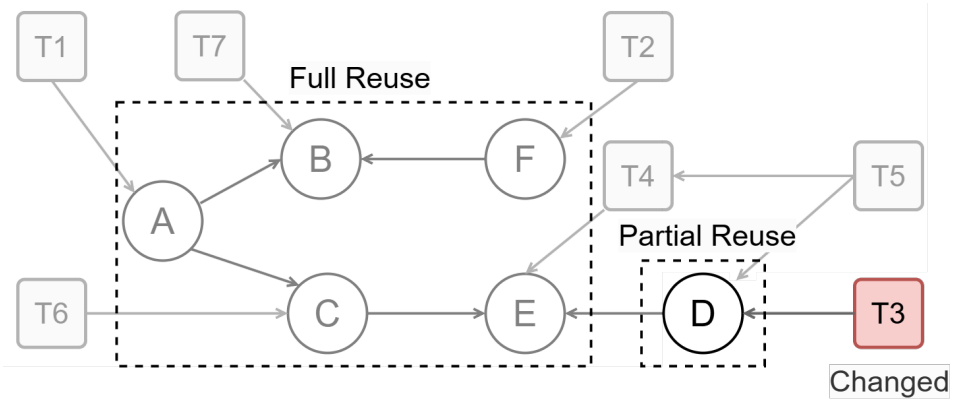
The mutation score resulting from regression mutation testing only accounts for some mutant-test execution pairs. In order to state the adequacy of the entire test suite, ReSuMo must integrate the partial testing results of revision R′ with the complete testing results of the prior revision R. To this end, ReSuMo accesses the historical mutation testing results of revision R, which were saved as a *baseline* in the same format as Listing 7.4. Each mutation object includes all the information necessary for the integration process, including its *killers* and *nonKillers*. Starting from the baseline of revision R, ReSuMo determines which mutant-test execution pairs can be carried over to the next revision, R′. In particular, a mutation m that was generated from a contract c and tested by test files $\{t_1, ...t_n\}$ is eligible for reuse only if the same contract c can produce m again in revision R′. Depending on the type of the changes between R and R′, ReSuMo distinguishes two types of reuse:

```
{
  ID: "dfa72ccc",
  file: ".../Contract.sol",
  start: 1295,
  end: 1299,
  startLine: 35,
  endLine: 35,
  original: " += ",
  replacement: "  = ",
  operator: "AOR",
  status: "killed",
  killers: [".../test/TestFile1.js"],
  nonKillers: [".../test/TestFile2.js"],
  testingTime: 17334
}
```

Listing 7.4: Example of Mutation Object generated by ReSuMo

Full Mutant Reuse: This scenario applies when contract c itself has not changed between R and R′, and none of the tests that exercise c have been modified or selected for re-evaluation. In this case, the mutation object m (including its status, *killers* and *nonKillers*) is reused in its entirety. It is treated as if it had been retested in R′, allowing ReSuMo to incorporate it directly into the new mutation testing results without additional checks. To illustrate this scenario, consider the dependency graph shown in Figure 7.3. This graph shows that only test file `T3` has changed in R′ with respect to the prior revison R. In such case, ReSuMo will generate all mutants from contract `D`, and re-execute them against `T3`. All mutants besides those of contract `D` can be fully reused.

Figure 7.3: Example of Full and Partial Mutant Reuse scenarios



Partial Mutant Reuse: This happens when m can be generated from c again, but only some of the tests that exercise c must be re-evaluated. For instance, if one or more test files have changed or if a new test file appears for c, ReSuMo cannot fully reuse m. New or modified tests must still be executed, and their outcomes are registered for m. Instead, results from the test files that were not re-evaluated are carried over and used to

update the information of m (including *killers*, and *nonKillers*, and its final *status*). To illustrate this scenario, consider again the dependency graph shown in Figure 7.3. Both test files `T5` and `T3` target contract `D`, but only `T3` has changed in R′. ReSuMo will only re-execute `T3` on the mutants of `D`. Listing 7.5 shows an example of partial results obtained for a mutant of contract `D`. As can be seen, `T3` did not kill the mutant, which is marked as live. When integrating these partial results with the full results from revision R (Listing 7.6), we find that the mutant was actually killed by the unchanged file `T5`, thus its status is updated to killed.

```
{
  ID: "mrab7dcd",
  file: ".../D.sol",
  status: "live",
  killers: [],
  nonKillers: ["./T3.js"],
}
```

Listing 7.5: Partial results for a Mutant of Revision R'

```
{
  ID: "mrab7dcd",
  file: ".../D.sol",
  status: "killed", //Updated
  killers: ["./T5.js"], //Reused
  nonKillers: ["./T3.js"],
}
```

Listing 7.6: Full results for a Mutant of Revision R'

7.4 Design and Implementation of ReSuMo

This Section presents the design and implementation of ReSuMo, an extension of SuMo that enables the regression mutation testing (RMT) workflow described in Section 7.3. ReSuMo is an open-source tool available on GitHub[2]. The following discussion details its high-level design, which is presented in Section 7.4.1, followed by an overview of its workflow in Section 7.4.2.

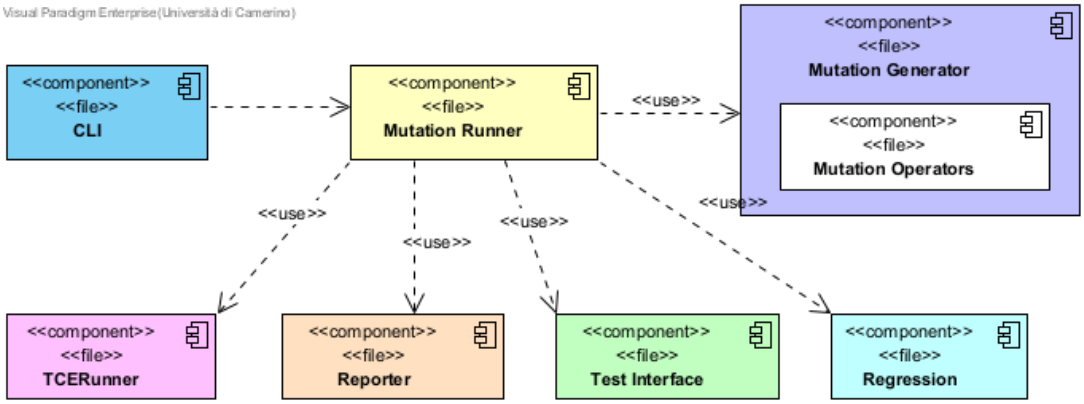
7.4.1 High-Level Design

ReSuMo operates within a NodeJS environment, enabling regression mutation testing for Solidity projects. While it retains the core architecture of SuMo, it introduces a novel logical module, *Regression*, which manages the file selection process described in Section 7.3. Additionally, several pre-existing modules have been modified to incorporate regression mutation testing capabilities. The high-level architecture of ReSuMo, including its components and interactions, is depicted in Figure 7.4. Since the majority of components have been discussed in Chapter 6, we focus here on the key modifications and additions.

Regression Module The *Regression* component implements the static analysis capabilities of ReSuMo. It executes key functionalities such as computing file dependencies, constructing a dependency graph, and generating dependency mappings. This module also identifies smart contracts that require mutation and determines which test files should be re-executed upon modifications to the project under test.

[2]ReSuMo repository: `https://github.com/MorenaBarboni/ReSuMo`

Figure 7.4: Components of ReSuMo



Reporter Module Similar to SuMo, the *Reporter* module in ReSuMo provides insights into the mutation testing process. However, in ReSuMo, it additionally manages the integration of *partially* and *fully reused* mutants, as detailed in Section 7.3.6. The *Reporter* module fulfills two primary roles:

- **Extraction of Test Outcomes:** During mutation testing, the *Reporter* gathers information on individual test executions and enriches each *mutant* object with the list of *killers* and *non-killers*. To track individual test outcomes, the project under test must integrate *Mochawesome*, a widely adopted JavaScript test reporter for frameworks such as Truffle and HardHat. ReSuMo parses the structured JSON reports generated by Mochawesome to extract file-level testing data.

- **Integration of Mutation Testing Results:** At the conclusion of mutation testing, the *Reporter* module inspects the mutation baseline from the prior revision (if available) to determine which mutation objects can be partially or fully carried over. It then integrates them with the partial results of the current revision to produce complete mutation testing results.

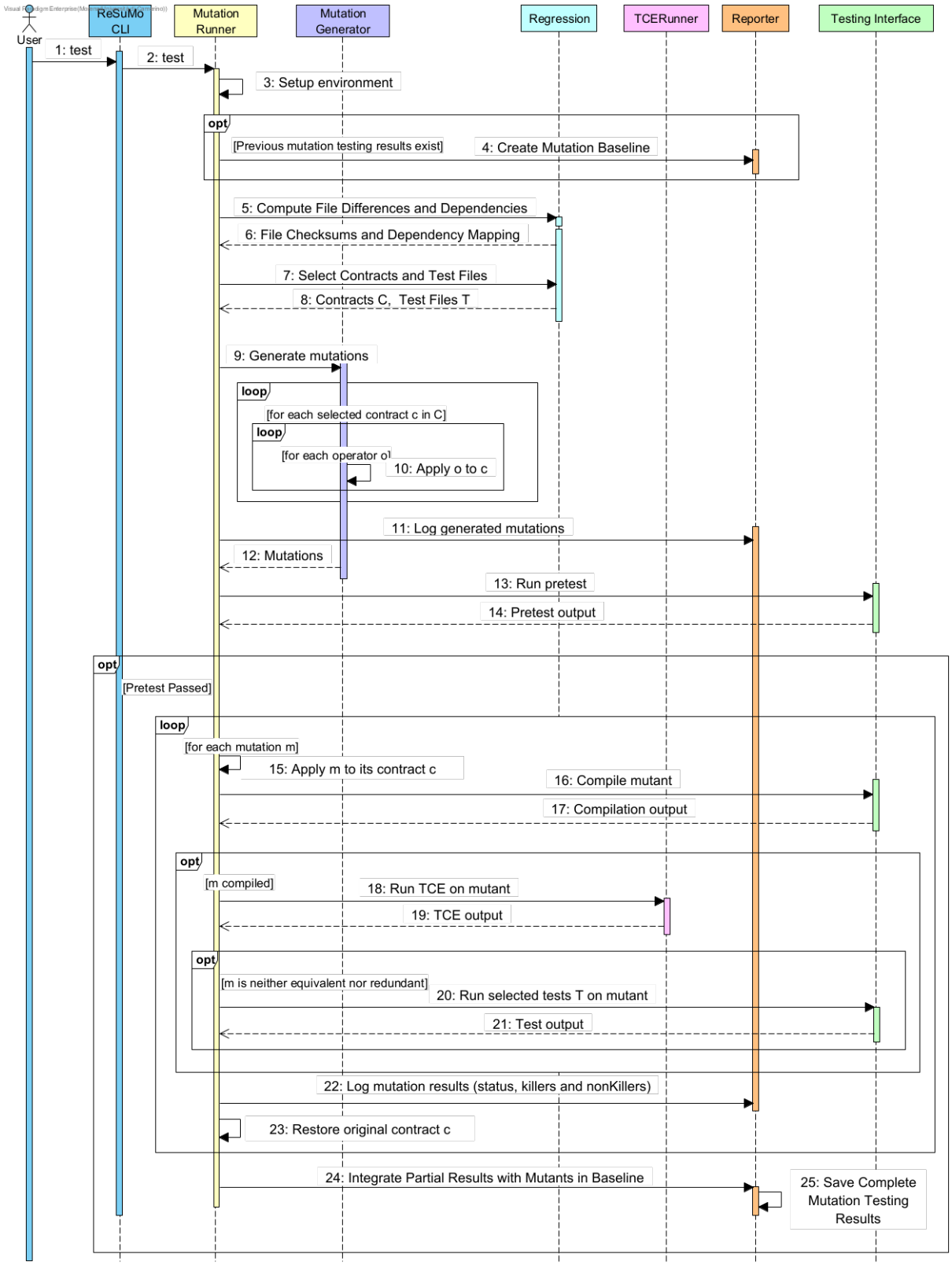
Currently, ReSuMo is a proof-of-concept tool that supports regression mutation testing for Truffle and HardHat-based Solidity projects. This is in part due to the heterogeneous languages and test reporters available across different testing frameworks, which require some ad-hoc modifications. In particular, the static analysis module currently works on JavaScript and Solidity test cases. Nevertheless, in the future we plan to extend ReSuMo to also support Foundry and Brownie regression mutation testing.

7.4.2 Regression Mutation Testing Workflow

The sequence diagram in Figure 7.5 illustrates the interactions between the components of ReSuMo when regression mutation testing begins. While the process follows many of the steps inherited from SuMo, we focus here on those central to ReSuMo's workflow. The key steps are summarized as follows:

Creation of the Mutation Baseline: At the start of the process, ReSuMo checks for existing mutation testing results from a prior revision, R. If such results are found,

Figure 7.5: Regression Mutation Testing process of ReSuMo



they are stored as a *mutation baseline*, a structured JSON file containing all mutation objects generated and tested in the previous execution.

Computation of File Differences and Dependencies: Once the baseline is created, the *regression* module identifies any modifications in the project directory since R. The *checksum* of all files in the current revision, R′, is also recorded for future mutation testing runs. Additionally, the *regression* module statically analyzes the codebase to construct a dependency graph and extract a *dependency mapping* that captures relationships among smart contracts and test files.

Selection of Contracts and Test Files: Based on the computed file differences and dependency structures, the *regression* module applies its selection strategy to determine which smart contract files must be mutated and which test files must be executed.

Mutant Generation: The process of mutant generation mirrors the one of SuMo, with the key distinction that only the contract files selected by the *regression* module are considered as input for mutation.

Mutation Testing Execution: Mutation testing is performed similarly to SuMo, however but ReSuMo leverages the *dependency mapping* to determine which test files cover each mutated contract. If no test file is associated with a mutated contract, the corresponding mutant is automatically classified as *live*. Otherwise, ReSuMo passes the list of required tests to the testing interface. Once testing is complete, ReSuMo updates the status of each mutant accordingly.

Mutant Logging: Following execution, ReSuMo updates the relevant mutation object with test results. If a mutant has been successfully tested (i.e., classified as either *live* or *killed*), the *reporter* module extracts details from the structured report generated by *mochawesome*. It then enriches the mutation object with information about the corresponding *killer* and *non-killer* test files.

Integration of Results: Once mutation testing is completed, the *reporter* consolidates the partial results from the current revision, R′, with the full set of results stored in the *mutation baseline* from R. The final, integrated results from R′ then serve as the baseline for subsequent regression mutation testing cycles.

7.5 Experiment Methodology and Setup

To evaluate ReSuMo we ran both full and regression mutation testing on consecutive revisions of three open-source Solidity projects, comparing their time costs and achieved mutation scores. In Section 7.5.1 we describe the selected projects, while the experimental setup is discussed in Section 7.5.2.

7.5.1 Subjects

To evaluate ReSuMo we selected three Solidity projects from GitHub. Since we planned to simulate the usage of ReSuMo during the evolution of the codebase, we selected four subsequent revisions of each project, $R_1, ...R_4$, resulting from recent commits (at the time of writing). In order to find suitable projects for the experiment, we defined the following selection criteria:

1. The selected applications must exhibit variability in terms of file complexity and the time required for testing. Although regression testing is typically applied in scenarios where the testing effort is substantial, evaluating ReSuMo on a diverse set of projects allows us to explore its effectiveness across different contexts.

2. Each project must include at least three commits that modify at least one smart contract or a test file, enabling us to simulate real-world software evolution. Furthermore, for these commits, the test suite must be executable without producing any failures.

3. The last requirement is purely technical and related to some constraints of the current implementation. In particular, the applications must include Javascript and/or Solidity test files, as other languages (e.g., Typescript) are not yet supported by ReSuMo.

Considering these criteria, we selected three projects from GitHub, which are summarized in Table 7.2. *Liquidity Protocol*[3] is an automatic market maker allowing digital assets to be traded in a permissionless and automatic way. *Farming*[4] offers protocols to give rewards to users holding specific tokens. *Limit Order Protocol*[5] allows users to place limit orders that later could be filled on-chain. For each project, we identified four subsequent project revisions, $R_1, ...R_4$, such that at least one meaningful project file (a smart contract or a test file) was modified since the previous revision.

Table 7.2 provides details about the total number of contracts (#C) and test files (#T) included in each project, as well as the number of contracts (ΔC) and test files (ΔT) that changed on each revision (R) considered. Note that the metrics only refer to a subset of contract and test files, in particular, libraries, interfaces, migrations, and mocks were excluded from the evaluation. Each subsequent project revision is identified by the relative GitHub commit hash.

7.5.2 Methodology

To answer the research questions posed in Section 7.2, we conducted a two-step experiment designed to measure both the efficiency of ReSuMo and the precision of its mutation testing results. The experiment follows a typical workflow where developers push commits to the project's codebase, and each new project revision undergoes mutation testing to ensure continuous test quality.

Methodology for RQ1: The objective of RQ1 is investigating the benefits ReSuMo brings to continuous mutation testing in terms of time reduction. To address RQ1, we performed:

1. *Full Mutation Testing (FMT):* As a baseline, we first executed SuMo on each revision of the selected projects. This involved generating mutants for all smart contracts and executing the corresponding test suites in their entirety.

[3]Liquidity Protocol repository: `https://github.com/1inch/liquidity-protocol`

[4]Farming repository: `https://github.com/1inch/farming`

[5]Limit Order Protocol repository: `https://github.com/1inch/limit-order-protocol`

Table 7.2: Experiment Subjects with Details about the Selected Revisions

DApp	R	#C	#T	ΔC	ΔT	Commit Hash
Liquidity Protocol	R_1	12	6	-	-	c31a03129b7d7c140bd2fc67c50a844998edd36c
	R_2	12	6	0	1	4ada0d41111339b84f67588f7116634064a2dbb7
	R_3	12	6	1	0	646cacedeac1d2689fa98f574a00eccd3cf6c96a
	R_4	12	6	1	0	2bf22dbc6301f625496118f7c5572ce5c28796a4
Farming	R_1	4	3	-	-	d97986ca7bf9bbed8181b465fff307f1b4f14a4e
	R_2	4	3	1	1	24cf89c533196e9e280d71e6fb18bbdc2c003c63
	R_3	4	3	1	0	3962ae1e0c58062e17f83392c030d798344db26c
	R_4	4	3	1	0	beae781549918faf42ff339b622a2354e0200803
Limit Order Protocol	R_1	6	6	-	-	70a41b7d7cca4aa321257c8d4fd2140d1b95f27f
	R_2	6	6	2	1	0c0036df78799680d932b7c097de7d1f23cb63a4
	R_3	6	6	2	1	bb57d3fd0cc3d06c873502f39cb220264198a5ff
	R_4	6	6	1	0	1ba21361aedfff942ba32f6f841314ad1c8061e8

2. *Regression Mutation Testing (RMT):* We then ran ReSuMo on the same revisions, allowing it to selectively analyze and mutate only those files that had been impacted since the previous revision.

To ensure a fair comparison, we used the same set of mutation operators and testing configurations for both the *full* and *regression* mutation testing runs. This ensured that any differences were attributable to the regression mutation testing strategy itself. Then, we measured the *total run-time* required for mutation testing in both the FMT and RMT scenarios. A significant reduction in run-time would indicate that ReSuMo is a viable alternative to FMT, making continuous testing more practical for developers.

Methodology for RQ2: To address RQ2, we examined the precision of the mutation testing results produced by ReSuMo relative to SuMo. In particular, for each project revision, we compared the mutation scores obtained from the FMT and RMT runs. Observing consistent mutation scores would confirm that the proposed RMT strategy is safe and can provide a reliable measure of test adequacy.

7.6 Discussion of Results

In the following, we discuss the results of the experiment presented in Section 7.5. We start with a general discussion of the Full and Regression Mutation Testing runs on the projects. Then, Section 7.6.1 and Section 7.6.2 present the answers to RQ1 and RQ2.

Full Mutation Testing In the FMT run, we executed SuMo on the selected project revisions $R_1, ...R_4$. For each revision R_n, Table 7.3 reports the total number of generated mutants $\#M$, uncompilable mutants $\#S$, timed-out mutants $\#O$, discarded equivalent or redundant mutants $\#D$, and successfully tested mutants $\#V$. Additionally, it provides

the mutation score (MS) achieved by the project's test suite and the total runtime required to complete the mutation testing process.

Table 7.3: Results of Full Mutation Testing using SuMo

DApp	R_n	$\#M$	$\#S$	$\#O$	$\#D$	$\#V$	**MS** (%)	**Time** (hh:mm:ss)
`Liquidity Protocol`	R_1	1673	179	4	318	1172	47,35	`11:57:31`
	R_2	1673	179	5	318	1171	47,31	`12:40:53`
	R_3	1674	179	6	318	1171	47,22	`12:01:37`
	R_4	1674	179	5	318	1172	47,27	`12:01:36`
`Farming`	R_1	369	42	0	112	215	63,26	`00:40:22`
	R_2	369	42	0	112	215	63,26	`00:40:22`
	R_3	374	42	0	113	219	63,01	`00:40:39`
	R_4	366	42	0	113	211	62,56	`00:39:47`
`Limit Order Protocol`	R_1	545	59	0	0	486	72,22	`03:13:31`
	R_2	545	59	0	0	486	72,63	`03:13:13`
	R_3	559	59	0	0	500	71,60	`03:18:43`
	R_4	558	59	0	0	500	71,14	`03:18:31`

Legend:
$\#M$: number of generated mutants
$\#S$: number of uncompilable mutants
$\#O$: number of timed-out mutants
$\#D$: number of equivalent or redundant mutants discarded by the TCE
$\#V$: number of valid mutants under test

Performing FMT on the four consecutive revisions proved to be computationally expensive. For a smaller project like *Farming*, the complete execution required approximately 2 hours and 40 minutes. Given that a single FMT run takes about 40 minutes, it is still feasible to conduct multiple evaluations daily. However, for *Liquidity Protocol*, each revision took around 12 hours to process, for a total of almost 49 hours. In such a case, it is not possible to get the FMT results back overnight, let alone perform frequent evaluations. Furthermore, we can observe that the MS of each project decreases as the codebase evolves. This trend is particularly interesting because certain commits in this study also modified test files (see Column 6 of Table 7.2). This suggests that the test suite is not evolving at the same pace as the codebase, potentially leaving parts of the modified code inadequately tested. In such scenarios, continuous mutation testing could provide valuable feedback, helping developers to proactively improve their test code.

Regression Mutation Testing In the RMT run, we executed ReSuMo on the selected project revisions $R_1, ...R_4$, allowing it to select a subset of test files and smart contracts for evaluation. Table 7.4 presents the file selection outcomes for each project under test. For each revision R_n, the Table reports the total number of contract files $\#C$ and test files $\#T$ in the project. $\#C'$ and $\#T'$ are the number of contract and test files that changed since the prior revision $R_{n\text{-}1}$. Based on these changes, ReSuMo selected $\#C_{mut}$ smart contracts to be mutated, and $\#T_{eval}$ test files to be evaluated. In particular, $\#T_{eval}$ is the total number of test files selected by ReSuMo for RMT at a global level.

However, the number of test files actually executed for each mutant might be lower. This is because each mutant is tested only with the subset of test files that explicitly import its corresponding smart contract as a dependency, as explained in Section 7.3.

Table 7.4: File selection output of ReSuMo for each project revision

DApp	R_n	$\#C$	$\#T$	$\#C'$	$\#T'$	$\#C_{mut}$	$\#T_{eval}$
`Liquidity Protocol`	R_1	12	6	-	-	12	6
	R_2	12	6	0	1	4	1
	R_3	12	6	1	0	4	1
	R_4	12	6	1	0	4	1
`Farming`	R_1	4	3	-	-	4	3
	R_2	4	3	1	1	3	2
	R_3	4	3	1	0	3	2
	R_4	4	3	1	0	3	1
`Limit Order Protocol`	R_1	6	6	-	-	6	6
	R_2	6	6	2	1	6	6
	R_3	6	6	2	1	6	6
	R_4	6	6	1	0	3	6

The selection strategy employed by ReSuMo appears to be particularly effective for the largest project, *Liquidity Protocol*. Across all revisions, we observe a substantial reduction in the number of smart contracts subjected to mutation testing, decreasing from 12 to 4, while the number of test files evaluated dropped from 6 to 1. The optimization is also evident in the *Farming* project, where, from revision R_2 onward, ReSuMo reduced the number of smart contracts mutated from 4 to 3, along with a similar decrease in the number of test files requiring re-execution. Instead, the results for *Limit Order Protocol* show a different trend. Here, the revision changes consistently led ReSuMo to select the same number of contracts and test files across most revisions, with the exception of R_4, where three smart contracts were excluded from the mutation process. Table 7.5 summarizes the results of RMT using ReSuMo. For each project revision, the mutant data reflects only the subset of contracts $\#C_{mut}$ chosen by ReSuMo for mutation testing. The table also highlights the total number of reused mutants $\#U$. As discussed in Section 7.3.6, these are the *fully reused* mutants carried over from the *mutation baseline* of the previous revision without being tested. Additionally, some test data for *partially reused* mutants may have been leveraged to determine the status of valid mutants in V, speeding up the mutation testing process. In the following Sections, we analyze and compare the results of the FMT and RMT runs to answer the research questions posed in Section 7.2.

7.6.1 Efficiency of ReSuMo

RQ1 aims to assess whether ReSuMo can reasonably reduce the time cost of full mutation testing in evolving smart contract projects. To answer this question, we compare the run-time of each RMT run with ReSuMo to the corresponding FMT run with SuMo. The results of this comparison are illustrated in Figure 7.6.

Table 7.5: Results of Regression Mutation Testing using ReSuMo

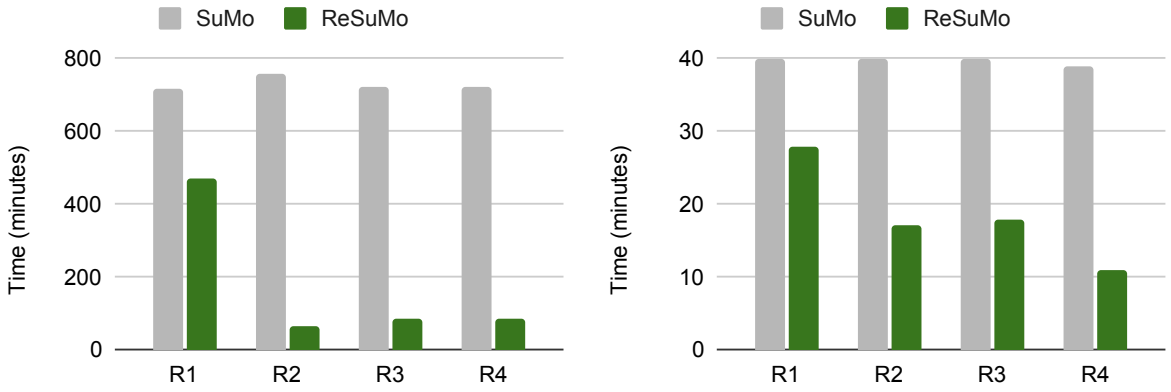
DApp	R_n	$\#M$	$\#S$	$\#O$	$\#D$	$\#T$	$\#U$	**MS** (%)	**Time** (hh:mm:ss)
Liquidity Protocol	R_1	1673	179	0	318	1176	-	47,62	7:50:51
	R_2	406	20	0	19	366	806	47,62	1:05:05
	R_3	1022	124	0	277	621	555	47,58	1:25:10
	R_4	1022	124	0	277	621	555	47,58	1:25:24
Farming	R_1	369	42	0	112	215	-	63,26	00:28:43
	R_2	234	22	0	109	103	112	63,26	00:17:26
	R_3	239	22	0	110	107	112	63,01	00:18:01
	R_4	255	30	0	112	113	98	62,56	00:11:24
Limit Order Protocol	R_1	545	59	0	0	486	-	71,6	03:12:21
	R_2	545	59	0	0	486	0	70,99	03:04:13
	R_3	559	59	0	0	500	0	71,20	03:12:25
	R_4	220	18	0	0	202	297	71,74	01:17:57

Legend:
$\#M$: number of generated mutants
$\#S$: number of uncompilable mutants
$\#O$: number of timed-out mutants
$\#D$: number of equivalent or redundant mutants discarded by the TCE
$\#T$: number of valid mutants under test
$\#U$: number of fully reused mutants

For the initial revision R_1, ReSuMo consistently reduced execution time across all projects. At this stage, no prior mutation testing results were available for reuse. The time savings were entirely attributed to the use of statically collected *program dependencies* to select the relevant test files for each mutated contract. Yet, leveraging dependency analysis alone allowed RMT to save about 4 hours in *Liquidity Protocol* (a $\approx$ 34.4% reduction, Figure 7.6a) and about 11 minutes in *Farming* (a $\approx$ 28.8% reduction, Figure 7.6b). Instead, *Limit Order Protocol* did not benefit from the availability of dependencies (a $\approx$ 0.6% reduction, Figure 7.6c), likely due to the higher coupling between project files.

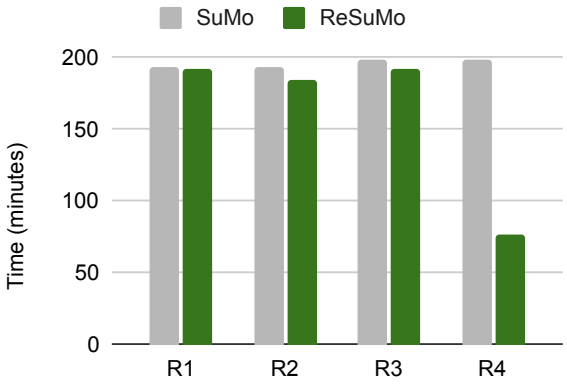
From revision R_2 to revision R_4 we can observe the benefits of the test and contract selection strategy implemented by ReSuMo. In *Liquidity Protocol*, regression mutation testing provided substantial time savings across all revisions. Between R_1 and R_2, only a single test file was modified, allowing ReSuMo to *fully reuse* 806 mutants from the previous *mutation baseline*. Since these mutants did not require re-evaluation, RMT completed in about one hour, compared to the 12 hours of FMT. In revisions R_3 and R_4 the changes impacted a smart contract file, reducing the number of fully reusable mutants. However, ReSuMo still achieved similar performance, saving over 10 hours of time in both cases. In this project, ReSuMo reduced the end-to-end run-time from R_1 to R_4 by $\approx$ 75.8%, and the average run-time for new revisions by $\approx$ 89.3%, making frequent re-evaluation possible.

While RMT is particularly beneficial for complex projects, it also proved valuable for smaller ones. In *Farming*, one FMT run only takes about 40 minutes, thus saving time in this scenario is not as critical. But since ReSuMo introduces minimal overhead during program analysis, it was still able to reduce the end-to-end run-time from R_1 to R_4 by



(a) Liquidity Protocol

(b) Farming



(c) Limit Order Protocol

Figure 7.6: Run-time of RMT (using ReSuMo) and FMT (using SuMo) on successive revisions of the projects under test

≈ 53.1%, and the average run-time for new revisions by ≈ 61.2%.

Instead, *Limit Order Protocol* is the project that benefited the least from RMT. This outcome was expected, as its revisions involved the largest number of directly impacted contracts and test files (see Table 7.4). Moreover, its highly interconnected dependency graph meant that a single file modification could trigger the selection of many test files for re-evaluation and numerous contracts for mutation testing. Therefore, in R_2 and R_3, the time savings were negligible and primarily attributed to the test selection based on dependency information. Instead, revision R_4 only features one changed smart contract. Here, RMT allowed to save about 2 hours of time, a ≈ 60% reduction relative to FMT. For this project, ReSuMo reduced the end-to-end run-time by ≈ 17.5%, and the average run-time for new revisions by ≈ 23%.

Across all projects, the results confirm that ReSuMo can significantly reduce the run-time of full mutation testing. However, they also highlight two key factors that influence these savings: 1) the *structure of the project* and 2) the *dispersion of code changes*.

ReSuMo is most effective in modular projects where test and contract files are loosely coupled. When each test file has a clear, contract-specific focus, the scope of re-evaluation

remains narrow, leading to substantial time savings. Instead, disorganized test files that cover multiple contracts tend to broaden the span of selected files, which reduces the efficiency of regression analysis. As shown in Table 7.4, when *Liquidity Protocol* evolves from R_1 to R_2, only one test file is modified. Since that file imports just four out of twelve smart contracts, the number of generated mutants remains limited, enabling faster execution. Projects with tightly coupled files are less likely to benefit from ReSuMo, as changes often propagate through the dependency graph and require broader re-analysis. For example, modifications to central contracts in densely interconnected systems (such as *Limit Order Protocol*) tend to trigger the selection of many related contracts and test files, diminishing potential time savings.

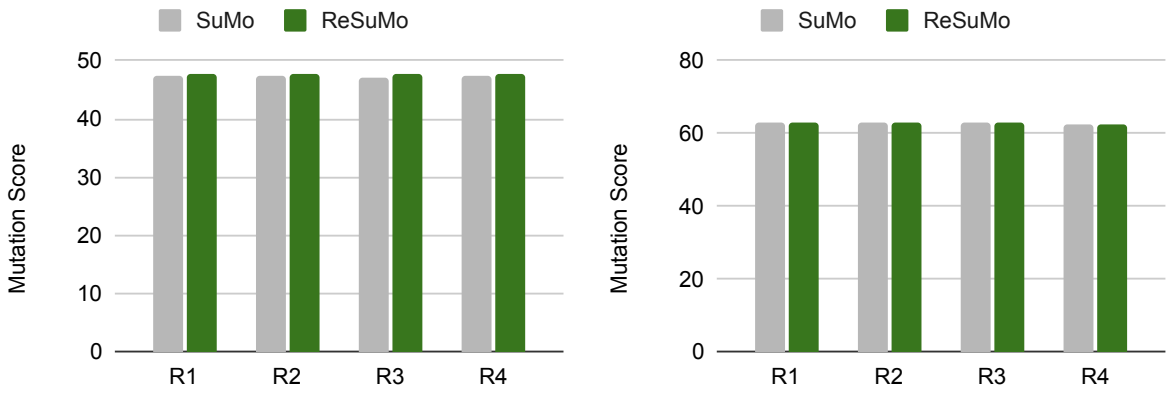
Equally important is the structure of the development process. ReSuMo is most effective when changes, whether introduced through one or multiple commits, remain localized to the same smart contract or test files. Indeed, scattered modifications across many unrelated files can trigger broad, and in some cases full, re-testing. This effect is evident in revisions R_2 and R_3 of *Limit Order Protocol*, where dispersed changes significantly reduced ReSuMo's efficiency. To maximize its benefits, developers should apply ReSuMo with an awareness of change locality.

> **Answering RQ1:** *Can ReSuMo reduce the time cost of Full Mutation Testing in evolving smart contract projects?*
>
> ReSuMo can significantly speed-up adequacy assessment in evolving smart contract projects. The proposed Regression Mutation Testing approach enables an average time savings of ≈ 48% per new project revision compared to Full Mutation Testing.

7.6.2 Safety of ReSuMo

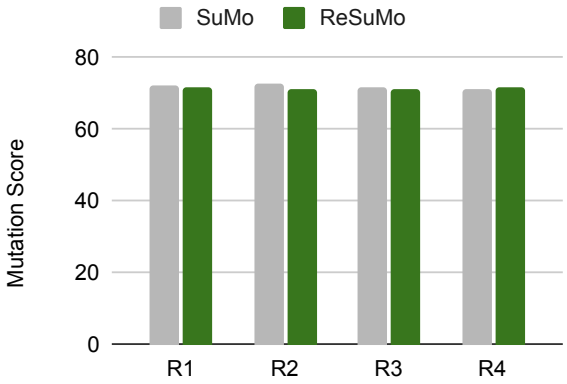
RQ2 aims to establish if ReSuMo is safe and can provide precise adequacy feedback to users despite its selective approach. Ideally, the final status (e.g., *live* or *killed*) of each mutant should be the same regardless of whether FMT or RMT is applied. To answer RQ2, we compare the mutation score produced by SuMo and ReSuMo on each project revision. The results of this comparison are illustrated in Figure 7.7.

For the initial revision R_1, the mutation scores remained largely consistent between the FMT and RMT runs. This stability was expected, as R_1 lacks prior mutation testing results to be reused. However, comparing the results for this revision provides insight into the safety of ReSuMo's static dependency analysis. During RMT, ReSuMo always builds a dependency mapping to know which test files import, and therefore must be executed on, the mutated contracts. Inaccuracies in the collection of dependencies could allow mutants to survive testing erroneously. As shown in Figure 7.7, the mutation score on R_1 is not identical for the FMT and RMT runs. The *Liquidity Protocol* project (Figure 7.7a) achieved $MS = 47,62\%$ during RMT, against the $MS = 47,35\%$ of FMT. However, this minor difference is not caused by missed program dependencies. Table 7.3 shows that 4 mutants timed-out during FMT. These same mutants were successfully tested once the test suite was reduced in size for RMT, causing slightly dfferent values for the MS. Similarly, the *Limit Order Protocol* shows different values on R_1, with $MS = 71,6\%$ for RMT, and $MS = 72,22\%$ for FMT. Further investigation revealed that at least one



(a) Liquidity Protocol

(b) Farming



(c) Limit Order Protocol

Figure 7.7: Mutation Score produced by RMT (using ReSuMo) and FMT (using SuMo) on successive revisions of the projects under test

of the project's test methods[6] exhibited flaky behavior (producing inconsistent results across repeated runs despite no changes to the code or tests). This non-determinism explains the discrepancies in mutation scores and may persist unless subsequent commits introduce a fix.

From revisions R_2 to R_4, we observe the mutation scores achieved by the projects as ReSuMo incorporates mutant-test execution outcomes from prior runs. For *Liquidity Protocol* (Figure 7.7a), the mutation score is still precise despite being computed from many *fully reused* mutants (and possibly several *partially reused* mutants). Differences between FMT and RMT are solely caused by timed-out mutants that were successfully tested in the RMT runs. By increasing the testing timeout value, these mutants would be successfully evaluated during the full runs as well, leading to identical mutation scores. For the *Farming* project (Figure 7.7b), RMT consistently produced mutation scores identical to FMT. Lastly, for *Limit Order Protocol* (Figure 7.7c), slight fluctuations in mutation scores were observed throughout revisions R_2 to R_3. Since all mutants were successfully tested during the full runs, these inconsistencies can once again be attributed to the

[6]Flaky method: `https://github.com/1inch/limit-order-protocol/blob/0c0036df78799680d932b7c097de7d1f23cb63a4/test/Interactions.js#L77`

non-deterministic behavior of the test methods included in the project. Such flakiness introduces variability unrelated to the RMT approach implemented by ReSuMo.

Overall, we can conclude that ReSuMo maintains a high degree of safety in regression mutation testing, producing mutation scores that are nearly identical to those obtained through full mutation testing. The observed discrepancies are not due to missed dependencies or errors in test selection but rather stem from external factors such as flaky tests and timeouts affecting mutant execution. This suggests that smart contract developers can confidently integrate ReSuMo into continuous testing workflows, in alternative to full mutation testing, without the risk of observing misclassified mutants.

> **Answering RQ2:** *Does ReSuMo produce precise mutation testing results in evolving smart contract projects?*
>
> ReSuMo embeds a safe Regression Mutation Testing approach and provides a precise measure of test suite adequacy. The Mutation Scores it produces match those from Full Mutation Testing, with minor discrepancies arising from external factors.

7.7 Threats to Validity

In this Section, we discuss the threats to the validity [147] of this study.

Threats to Construct Validity The construct validity reflects to what extent the operational measures that are studied represent what is investigated according to the research questions. One possible threat to construct validity is the way we envision the usage of ReSuMo in a real development setting. The idea behind regression mutation testing stems from the necessity of continuously re-evaluating the test suite of a given Solidity project. In this thesis, we considered the single commit as a checkpoint for the evaluation of the test suite. However, this decision should be adapted case by case, considering the rhythm of the development team and the extensiveness of the modifications. Moreover, we always selected recent commits as we envision the usage of ReSuMo while the project is somehow becoming more mature and complex. At an early stage is probably not worth engaging in the derivation of a comprehensive test suite, while it is generally important to keep its adequacy high after some iterations.

Threats to Internal Validity The threats to internal validity address how dependent the data is on the conducted research and whether the outcomes can be accurately reproduced. The presence of *flaky tests* in the test suites shipped with the selected projects constitutes the main threat to the validity of the results. Since flaky tests can exhibit non-deterministic outcomes they constitute a relevant problem for mutation testing [152]. Indeed, the mutation score achieved by a test suite might not be the same on two identical re-runs if such methods are present. To mitigate this risk, before choosing a project we made sure to run its test suite several times to possibly detect inconsistent outcomes. However, some flaky tests might be difficult to detect using a simple re-run strategy. This is the case for the *Limit Order Protocol* project, for which we discovered a flaky

test method only after completing the regression mutation testing runs and analyzing the experimental data.

Threats to External Validity The threats to external validity determine the extent to which our results can be generalized to other projects. To evaluate ReSuMo we selected a short sequence of commits for a set of open-source Solidity applications. Indeed, the experiment we designed has a rather high cost: for each project, we repeated both a full and a regression mutation testing run on each of the four revision. However, the time savings achieved by ReSuMo might vary depending on the project's internal structure, and on the dispersion of the changes introduced by each commits. To mitigate this threat, we choose projects with different sizes, and used real-world commits that impact both contracts and test files. Large-scale experiments should be conducted to have a definitive answer on the effectiveness of the approach.

7.8 Related Work

Despite the increasing adoption of mutation testing for Ethereum smart contracts, no work explores the intersection with regression testing in this domain. Existing studies relevant to our proposal focus on two key areas, mutation testing approaches for smart contracts, and regression mutation testing techniques for evolving software systems.

Mutation Testing for Ethereum Smart Contracts In recent years, several mutation testing frameworks and tools have been developed to assess the quality of Ethereum smart contract test suites. However, none of these incorporate a dedicated regression testing mechanism. Wu et al. [192] introduced *MuSC*, the first mutation testing framework for Solidity smart contracts. Their approach defined a compact set of Solidity-specific and JavaScript-oriented mutation operators to introduce faults into smart contract code. Honig et al. [76] proposed *Vertigo*, a prototype targeting Solidity contracts, while Andesta et al. [3] designed a novel set of mutation operators based on an analysis of known vulnerabilities in Ethereum contracts. These were integrated into the the *Universal Mutator* tool. Chapman et al. [40] presented *Deviant*, a comprehensive mutation testing tool featuring operators at four granularities: inter-module, intra-module, intra-function, and intra-statement. Their evaluation across three real-world projects demonstrated that *Deviant* could enhance Solidity application quality by exposing weaknesses in test suites. Hartel and Schumi [75] derived mutation operators inspired by the Mothra set, in addition to four Solidity-specific mutation operators, and implemented them in the *ContractMut* tool. They also introduced a novel mutation adequacy criterion based on deviations in gas consumption, providing an alternative to traditional test suite effectiveness metrics.

Mutation Testing for Evolving Systems The application of mutation testing to evolving systems introduces significant computational challenges. Traditional mutation testing requires rerunning an entire test suite against all generated mutants, which is prohibitively expensive in large-scale and continuously evolving projects. Zhang et al. first introduced Regression Mutation Testing with ReMT [200], an approach that accelerates mutation testing by incrementally computing mutation scores across software

revisions. Their method leverages control flow graph analysis to identify program regions where test behaviors remain unchanged, allowing for selective test execution. Cachia et al. [36] introduced incremental mutation testing, limiting mutant generation to modified code regions. Their findings confirmed a substantial reduction in test execution time without compromising fault detection capability. Chen et al. [42] explored the performance of different regression test selection (RTS) techniques to speed up mutation testing in evolving software systems. They conducted an extensive empirical study across large-scale Java projects and found that file-level static and dynamic RTS methods can significantly enhance mutation testing efficiency. A more recent contribution by Ma et al. [104] introduced commit-aware mutation testing, which evaluates the extent to which committed changes influence program behavior. Their experiments on Java and C projects revealed inefficiencies in traditional mutant selection strategies, which often fail to prioritize changes that introduce defects.

7.9 Directions for Future Research

This thesis presented ReSuMo, the first static and file-level regression mutation testing framework for Ethereum smart contracts. Future work should explore the usage of dynamic, method-level regression mutation testing. Using a finer-grained approach may further enhance efficiency, especially in small and tightly-coupled smart contract projects where file-level RMT offers limited advantages. Allowing developers to configure the level of computational granularity would make ReSuMo more adaptable to different testing needs.

On the same note, understanding how dynamic dependency collection impacts the efficiency and precision of ReSuMo is an interesting research direction. While static analysis provides a lightweight and scalable approach, dynamic techniques could offer finer-grained insights into test coverage and dependencies.

One problem is that Solidity testing frameworks are heterogeneous and generate coverage reports in different formats, often aggregating results without linking coverage information to specific test files (except in plugins like *solidity-coverage* for HardHat). To integrate dynamic dependency collection into ReSuMo, we are working on an agnostic coverage extraction mechanism capable of processing coverage reports from multiple frameworks (e.g., *Hardhat*, *Foundry*, and *Brownie*). This approach would provide a standardized interface for test-to-contract mappings.

PART III

MUTATION TESTING FOR SMART CONTRACT AUDITING

CHAPTER 8

MUTATION TESTING FOR SMART CONTRACT CODE INSPECTION

Smart contracts hold the potential to revolutionize various industries, but their implementation requires thorough testing due to the associated financial risks. Mutation testing is a powerful technique that can boost the fault-detection capabilities of a test suite, but it can also foster a deeper understanding of smart contract behavior. This Chapter investigates the productivity of mutants with respect to their capabilities in disclosing Solidity issues. Based on these findings, it proposes an enhanced mutation strategy to better assist smart contract auditors during code inspection activities.

This chapter is based on material from the following publication: [10]

* * *

8.1 Introduction

Smart contracts have the potential to revolutionize various industries by automating transactions and removing the need for intermediaries. However, the risks associated with their implementation necessitate thorough testing before real-world deployment [204, 21]. Even a minor coding oversight can lead to financial loss or exploitation by malicious actors. Recognizing these risks, specialized *auditing* companies have started offering services to review smart contract security and functionality. As discussed in Section 4.3, auditors typically conduct rigorous code inspections, run automated analysis tools (e.g., Slither [61], Echidna [70]), and evaluate the quality of the test suite provided by the development team.

While clients remain responsible for implementing fixes and improvements, auditors deliver a comprehensive report that recommends corrective and perfective actions. Within this process, *coverage analysis* benefits auditors in two ways. First, it allows them to evaluate the test suite submitted by clients and give them feedback to improve test quality. Second, it supports manual code inspection by revealing under-tested areas that

may contain overlooked faults or vulnerabilities. Still, coverage remains a coarse metric that cannot offer in-depth insight into the quality of test cases or their ability to disclose faults in the code under test [81].

Mutation testing is a test adequacy assessment technique that could support auditing in a more comprehensive manner. Research on mutation testing for blockchain-based applications already demonstrated the value of Solidity mutants in improving smart contract test suites [22, 40]. Furthermore, an industrial study conducted at Google [139] showed that analyzing live mutants sometimes prompted developers to introduce code fixes or design changes during the code review process. In a subsequent study, the authors analyzed over 1500 mutant-related merge requests at Google [138], shedding light on their value in code inspection and review. While test improvement emerged as the most common approach for resolving mutants, they also observed instances of code refactorings and in-depth discussions about the codebase. This suggests that the value of live mutants extends beyond test improvement alone.

Despite the central role of auditing in smart contract development, the potential for live mutants to support smart contract code inspection remains largely unexplored. Mutation analysis could offer a unique perspective to auditors, helping them to reason about the code, and in some cases even uncovering faults in the implementation. In this thesis, we explore the value that *mutation testing* can bring in the smart contract code review process. Specifically, we investigate whether inspecting live mutants might have any role in the disclosure of issues within Solidity smart contracts.

To investigate this aspect, we run mutation testing on a set of projects audited at *Quantstamp*[1], a leading blockchain security company. The empirical study suggests that mutation analysis can support smart contract auditing in multiple ways. Indeed, besides guiding clients in strengthening their test suites, mutation testing can sometimes surface issues in the code under test. Additionally, based on experimental results and auditor's feedback, we further refine the mutation operator implemented by SuMo. The results show a $\approx 30\%$ reduction in the number of generated mutants, while increasing the set of productive mutants related to issues by $\approx 43\%$ overall.

The rest of this Chapter is organized as follows. Section 8.2 describes the research objectives, while Section 8.3 presents the relative methodology. Section 8.4 discusses mutant productivity, focusing on the potential for live mutants to support smart contract code reviews. This is followed by the analysis, refinement and evaluation of the mutation operators in Section 8.5. Section 8.6 presents the threats to the validity of this study, while Section 8.7 outlines related work. Lastly, Section 8.8 identifies areas for future research.

8.2 Research Objectives

In this study, we aim to explore the role of mutation testing in smart contract auditing and investigate how mutation operators can be refined to better support auditors in their review process. The following objectives guide our work:

[1]Quantstamp: `https://quantstamp.com`

1) Investigating the role of Mutation Analysis in Smart Contract Auditing. The first objective of this study is to assess the potential of mutation analysis as a complementary technique in smart contract auditing. While mutation testing is widely recognized for its ability to enhance test quality, its applicability in manual code inspection remains underexplored. Through this objective, we aim to determine whether live mutants, when surfaced during manual code review, can provide actionable insights to auditors. Specifically, we seek to understand whether such mutants directly reveal flaws in the smart contract under review, assisting auditors in their assessment. To achieve this, we conduct an empirical analysis of Solidity projects previously audited by Quantstamp, a leading blockchain security company. Mutation testing is applied to the original versions of these projects, followed by a retrospective examination of audit reports to identify cases where mutation analysis could have contributed to issue detection.

2) Optimizing Mutation Operators to Better Support Smart Contract Auditors. The second objective of this study is to refine and enhance mutation operators to improve their utility in smart contract audits. While mutation testing can surface weaknesses in test suites, its impact is heavily influenced by the quality and relevance of the mutation operators used. Some mutation operators may generate uninformative mutants, increasing cognitive overhead for auditors without yielding significant insights. Therefore, this objective focuses on identifying the most effective mutation operators for smart contract auditing and exploring ways to refine or introduce new operators that might be helpful during code review. To accomplish this, we collect qualitative feedback from experienced auditors. The auditors evaluate existing mutation operators in SuMo, providing insights into their perceived utility and suggesting improvements or additional mutation rules. Based on this feedback, we refine the mutation operators and reassess their effectiveness.

8.3 Methodology

This Section outlines the methodology we followed considering the research objectives presented in Section 8.2, followed by the presentation of the experimental subjects.

8.3.1 Analysis of Productive Mutants

The first objective of this study is to assess the potential of mutation analysis in smart contract auditing. To this end, we need to clarify what exactly makes a mutant *productive* from the auditor's perspective. Previously, Petrovic et al. [139] distinguished between productive and unproductive mutants based on the practical value of a mutant's impact. Specifically, a mutant is productive if it either elicits an effective test, or its analysis advances knowledge and code quality. This distinction is inherently qualitative, depending on the developer's judgment. Unlike developers however, auditors are not directly responsible for deriving new test cases. Thus, we reframe the concept of productive mutants to consider their role in the discovery of issue within the smart contract under review. Specifically, we define two categories of productive mutants:

- **Test-Productive Mutant (TPM)**: A mutant that aids in the creation of a test capable of uncovering an issue;
- **Inspection-Productive Mutant (IPM)**: A mutant that directly reveals the presence of an issue during code review.

Considering these definitions, IPMs are particularly relevant to auditors, as they might help them disclose issues in the contract under review without directly writing new test cases. To investigate mutant productivity, we run SuMo on a set of projects submitted to Quantstamp for a security audit. We execute mutation testing on the original versions of these projects as provided by clients. Subsequently, we analyze whether any live mutants could retrospectively have helped in the identification of issues found by auditors in the smart contracts. To this end, we obtained the audit reports from auditors, which documented all issues identified during the review. To focus on issues relevant to mutation testing, we exclude findings such as recommendations for documentation improvements or the incorporation of new logic due to specification changes. Then, for each Mutation Testing Relevant Issue (MTRI) in the audit reports, we perform the following steps:

1. Analyze the characteristics of the issue, how it impacts the program behavior, and the recommended fix;
2. Manually assess whether one or more live mutants produced by SuMo could have helped to disclose the issue;
3. Tag the issue based on its association with the following mutant categories:
   (a) **No Productive Mutants (NPM)**: no live mutants could have helped to disclose the issue;
   (b) **Test-Productive Mutants (TPM)**: one or more mutants could have helped to disclose the issue as a result of test improvement;
   (c) **Inspection-Productive Mutants (IPM)**: one or more mutants could have helped to disclose the issue as a result of direct mutant inspection.

Note that an issue might be associated with both Test and Inspection Productive Mutants. Of particular interest to us is the IPM category, for two reasons. First, the existence of IPMs can demonstrate the value of mutant inspection in smart contract auditing. Second, analyzing IPMs may reveal patterns and mutation operators that can directly aid in identifying issues during code reviews.

8.3.2 Enhancement of the Mutation Operators

The second objective of this study is to refine and enhance mutation operators to improve their utility in smart contract audits. To this end, we employed a three-phase approach:

1) Analysis of Productive Mutants: We examined the productive mutants generated by SuMo in relation to each mutation operator. We also inspected the issues for which

no productive mutants were generated in the previous experiment.

2) Auditor Feedback Collection: Recognizing that practitioner input is critical in defining heuristics for generating valuable mutants (as demonstrated by Petrovic et al. in Google's internal study [137]), we conducted informal one-on-one interviews with three experienced smart contract auditors from Quantstamp. The interview aimed to assess the utility of existing mutation operators and gather suggestions for enhancements. Each interview followed a format with two main objectives:

- *Evaluation of Existing Mutation Operators*: We presented auditors with each of the mutation operators implemented in SuMo. For each operator, we explained its transformation rule and intended purpose. Auditors provided qualitative feedback on whether they perceived each mutation as useful, confusing, or unnecessary.

- *Gathering Suggestions for New or Improved Operators*: After reviewing existing operators, auditors were asked open-ended questions to explore possible improvements. They suggested new mutation rules that could target common smart contract faults or vulnerabilities, particularly those that existing operators failed to capture. They also provided insights on how certain mutations might be adjusted to make them more practical or informative.

3) Evaluation of Improved Mutation Operators: We consolidated previous feedback, aiming to increase the proportion of mutants that contribute to meaningful code inspection. Following the auditor feedback, we updated the mutation operators of SuMo by 1) removing or modifying ineffective operators that auditors found unhelpful, and 2) introducing or updating operators based on suggestions. To measure the impact of these changes, we repeated the retrospective study described in Section 8.3.1. Specifically, we assessed whether the refined operators increased the number of IPMs, improving the direct utility of mutation testing for auditors.

8.3.3 Experiment Subjects

We conducted the experiments on 8 Solidity projects submitted to *Quantstamp* for an audit. To ensure a meaningful analysis, all projects include a test suite with good code coverage. In Table 8.1, we provide details about the projects supplied by *Quantstamp*. Due to the sensitive nature of the data, we identify each subject with a unique ID. We also report the number of contract files (CF), contract lines of code (LoC), and test methods (TM) in the codebase with the total statement coverage. Note that Project 3 displays a low value, but most of its contracts in scope for the audit were thoroughly covered by the tests. Lastly, we report the total number of issues disclosed by auditors, and those that are relevant for the analysis with mutation testing (Mutation Testing Relevant Issues, or MTRI), grouped by severity level (*Low*, *Medium*, *High*, or *Other*).

Table 8.1: Experiment Subjects

ID	LoC	CF	TM	Stmt. Cov.	Total Issues	#Low MTRI	#Medium MTRI	#High MTRI	#Other MTRI
1	2,105	14	83	96.9	21	2	4	-	-
2	564	11	141	90	29	4	4	2	2
3	3,102	42	155	57.4	25	3	4	2	-
4	1,815	19	96	98	48	9	5	6	2
5	1,678	10	118	91.4	26	4	1	3	-
6	641	10	165	98.8	11	3	3	-	-
7	68	1	11	82.6	7	3	-	-	-
8	2,745	21	339	75.6	18	8	-	1	4
Tot.	12,718	128	1,108	-	185	36	21	14	8

Legend:
LOC: Lines of Code, CF: Contract Files, TM: Test Methods, Stmt. Cov.: Statement Coverage, MTRI = Mutation Testing Relevant Issues

8.4 Analysis of Productive Mutants for the SuMo Mutation Operators

This Section discusses the results of the experiment described in Section 8.3.1. Table 8.2 shows the mutation testing results obtained using the original set of SuMo operators. For each project (ID), we reported the total number of *generated mutants* (M), *valid mutants* (V) that are not *uncompilable*, or *timed-out*, and the achieved Mutation Score (MS). The MS is the ratio of *killed* mutants to the total number of valid mutants.

Overall, SuMo generated and tested 18,779 mutants in about 302 hours of time. Although most audited projects were submitted with high-quality test suites, nearly all achieved low or mediocre Mutation Scores (≈ 62 on average). This result is a symptom of underlying weaknesses in the test suites. A low MS indicates that the test cases may not be as comprehensive or effective as intended. This observation was later confirmed when auditors reported tens of issues in the reviewed projects, validating the connection between low Mutation Scores and undetected code faults.

Generation of Productive Mutants For each project, Table 8.2 shows the number of Mutation Testing Relevant Issues (MTRI) related to No Productive (NPM), Test-Productive (TPM) and Inspection-Productive (IPM) mutants (as defined in Section 8.3). First, we can see that SuMo generated productive mutants for about 47% of the MTRI documented by auditors. Most of these (≈ 32%) are related to TPM*s*, that is, issue disclosure is dependent on the derivation of a test case capable of killing a surviving mutant. Considering that auditors are not responsible for writing test cases, such mutants are not of particular interest to them. However, it certainly confirms the value of this technique for customers. If test cases were derived for such live mutants, they might have been able to disclose the issues in the first place.

Among the analyzed smart contract issues, a small percentage (≈ 15%) feature mutants that could have provided insights during code inspection. Listing 8.1 shows an

Table 8.2: Analysis of Productive Mutants Generated by SuMo

ID	#M	#V	MS	Total MTRI	With NPM	With TPM	With IPM	With both TPM & IPM
1	3,177	2,720	55.6	6	4	2	0	0
2	727	622	63.8	12	8	4	0	0
3	3,836	3,077	64.3	9	7	2	0	0
4	2,069	1,791	51	22	11	8	1	2
5	2,743	2,320	69.3	8	3	2	2	1
6	922	692	81.5	6	3	2	1	0
7	134	117	69.2	3	2	1	0	0
8	5,171	4,589	43.3	13	4	4	1	4
Tot.	18,779	15,928	-	79	42	25	5	7

Legend:
M: Generated Mutants, V: Valid Mutants, MS: Mutation Score, MTRI: Mutation Testing Relevant Issues, NPM: No Productive Mutants, TPM: Test-Productive Mutants, IPM: Inspection-Productive Mutants

```
1 function _getData(...) internal returns(...) {
2 ⊙  while (latestPrice <= 0) { ... }
3 ⋆  while (latestPrice < 0) { ... }  {
4 }
```

Listing 8.1: Example of Inspection-Productive Mutant

example of such mutants. The `_getData` function includes an *Incorrect Validation* on data retrieved from an external oracle. Indeed, it relies on a price input (`latestPrice`) that may assume a zero value, a case that the smart contract does not account for. Here, SuMo injected a relational mutation that changed the faulty conditional `latestPrice <= 0` to `latestPrice < 0`. Upon further inspection, auditors may notice that this mutant survived because it actually fixed the code under test. In fact, inspecting the audit report revealed that this mutant is equivalent to the patch later applied by clients. While not all IP mutants necessarily serve as patches, in this instance, the live mutant could have disclosed and suggested how to address the issue.

Based on the experimental results, mutation analysis can support smart contract auditing in multiple ways. First, the high proportion of TPMs confirms that mutation testing can guide clients in strengthening their test suites. Even though auditors are not responsible for writing tests, mutation analysis can provide test improvement tips that might lead to issue detection. Most importantly, the existence of Inspection-Productive Mutants (IPMs) demonstrates that mutation testing can sometimes surface issues directly through mutant inspection. While only 15% of issues benefited from IPMs, this is a first confirmation of the potential value of mutant analysis.

Table 8.3: Category of Issues with Associated Productive Mutants

Category of Issue	Total MTRI	With NPM	With TPM	With IPM	With both TPM &IPM
Access Control	7	6	0	1	0
Aliasing	1	1	0	0	0
Block Variable Dependence	1	0	0	1	0
Checks-Effects-Interactions Violation	2	1	1	0	0
Gas usage	4	0	3	0	1
Integer Over/Underflow	1	1	0	0	0
Initialization	3	2	1	0	0
Event Log Usage	1	0	0	1	0
Miscalculation	11	3	6	0	2
Logical Oversight	4	1	2	0	1
Other	3	2	1	0	0
Precision Loss	4	1	1	1	1
Reentrancy	3	3	0	0	0
Timing	1	0	1	0	0
Unchecked Transfer	5	4	0	1	0
Validation	28	17	9	0	2
Tot.	79	42	25	5	7

Categories of Issues linked to Productive Mutants Overall, SuMo generated productive mutants at a rate of ≈50% for High, ≈37% for Medium, and ≈47% for Low Severity MTRI. Table 8.3 further breaks down the results with respect to each category of issue, revealing distinct patterns in the generation of productive mutants. Categories such as *Miscalculation*, *Logical Oversight*, *Precision Loss*, and *Missing or Incorrect Validation* include various related TP and IP mutants. These encompass issues commonly encountered in smart contracts. Instead, categories like *Access Control*, *Block Dependence*, *Reentrancy*, and *Unchecked Transfer* had limited or no productive mutants associated with them. These issues may prove more challenging to reveal using mutation testing, primarily due to their nuanced and complex nature. The ability of mutation analysis to generate productive mutants in certain issue categories suggests that mutation operators could be refined to improve their relevance in auditing contexts.

Key Insights: *Generation of Productive Mutants*

Mutation analysis could have helped to discover about 47% of the analyzed issues. Test-Productive and Inspection-productive mutants were found for about 32% and 15% of the findings, respectively. The latter can be particularly beneficial to auditors by offering direct insights during code inspection.

8.5 Analysis of Productive Mutants for the Enhanced Mutation Operators

In this Section, we first discuss how the SuMo operators performed on the audited projects. Based on such analysis, supported by practical feedback from auditors, we optimize the set by removing rules, improving the existing ones, or introducing new operators. In particular, Section 8.5.1 presents the operator enhancements, whereas Section 8.5.2 discusses the results of the retrospective study for these enhanced operators.

8.5.1 Enhancement of the Mutation Operators

For each SuMo operator, Table 8.4 shows the respective programming aspect (as defined in Chapter 6), the total ($\#M$) and valid ($\#V$) number of mutants it generated for all projects, and its Mutation Score. It also shows the number of issues for which that operator generated at least one Test-Productive (TPM) or Inspection-Productive (IPM) mutant. Operators that generated at least one productive mutant are highlighted in bold.

Control Structures

Mutations involving control structures generated productive mutants throughout the experiments and do not necessitate major revisions. Particularly, CSC (Conditional Statement Change), an operator that swaps each condition with `true` and `false`, indicated that certain conditional statements were not properly checked by the test suite. As a result, this operator generated IP and TP mutants for 1 and 5 issues (See Table 8.4). These span over various issue categories, but all of them could have been detected by a more thorough test suite. Less frequent control mutations, such as those targeting loops - LSC (Loop Statement Change) and BCRD (Break Continue Replacement and Deletion) - generated TP mutants as well, most of which related to inefficient code and *gas usage*.

Events

Mutations injected by EED (Event Emission Deletion) can help to ensure proper log verification by commenting out event emission statements in the contract. Indeed, the *incorrect event emission* reported by auditors (See Table 8.3) was effectively disclosed by EED. Additionally, auditors suggested the insertion of mutation operators for swapping indexed event parameters. When a Solidity event is emitted, indexed parameters are evaluated before non-indexed parameters, in right-to-left order. This unusual evaluation order might result in unexpected outcomes if the parameters have conflicting side effects[2].

Event Argument Swap (EAS): The new EAS operator implemented by SuMo looks for function calls within event emission statements and, if they correspond to indexed parameters, it swaps them to simulate a different evaluation order.

[2]Exploit example: `https://github.com/ethereum/solidity-underhanded-contest/blob/master/2022/submissions_2022/submission9_TynanRichards/SPOILERS.md`

Table 8.4: Results grouped by Mutation Operator

Aspect	ID	$\#M$	$\#V$	MS	MTRI with TPM	MTRI with IPM
Control Structures	**BCRD**	44	41	63.4	**1**	**0**
Control Structures	CBD	4	4	25	**1**	**0**
Control Structures	**CSC**	747	730	59	**5**	**1**
Control Structures	**LSC**	182	167	80.2	**3**	**0**
Events	**EED**	223	221	25.8	**4**	**1**
Function Parameters and Return Values	**RSD**	470	462	68.8	**6**	**0**
Function Parameters and Return Values	RVS	28	16	62.5	0	0
Inheritance (Constructor)	CCD	30	12	75	0	0
Inheritance (Overloading)	ACM	2	1	0	0	0
Inheritance (Overloading)	OLFD	17	3	33.3	0	0
Inheritance (Overriding)	SKD	31	29	69	0	0
Inheritance (Overriding)	SKI	85	3	33.3	0	0
Inheritance (Overriding)	ORFD	272	22	59	0	0
Modifiers (Custom)	MOC	38	37	2.7	0	0
Modifiers (Custom)	**MOD**	337	319	28.5	**5**	**0**
Modifiers (Custom)	**MOI**	608	494	52.6	**0**	**1**
Modifiers (Custom)	MOR	553	540	52.2	0	0
Modifiers (State Mutability)	PKD	13	3	0	0	0
Modifiers (Visibility)	FVR	1849	998	41	0	0
Modifiers (Visibility)	VVR	541	359	12	0	0
Operators	**AOR**	525	525	83	**3**	**0**
Operators	**BOR**	7738	7394	62.8	**19**	**3**
Operators	DOD	22	21	33.3	0	0
Operators	ICM	27	1	100	0	0
Operators	**UORD**	312	300	76.3	**1**	**0**
Libraries	SFR	52	52	5.8	0	0
Special Variables and Functions	**GVR**	500	412	59	**1**	**1**
Special Variables and Functions	**EHC**	655	623	47.4	**7**	**0**
Special Variables and Functions	**ETR**	64	40	65	**0**	**1**
Special Variables and Functions	TOR	119	119	34.4	0	0
Special Variables and Functions	MCR	92	87	67.8	0	0
Types (Conversions)	ECS	76	28	60.7	0	0
Types (Data Location)	DLR	556	64	20.3	0	0
Types (Data Types)	AVR	112	99	82.8	0	0
Types (Data Types)	**BLR**	169	169	65.7	**1**	**1**
Types (Data Types)	**ILR**	1375	1303	44.9	**12**	**4**
Types (Data Types)	ER	35	32	34.4	0	0
Types (Data Types)	SLR	200	130	60	0	0
Variable Units	**VUR**	76	68	69.1	**0**	**1**

```
1  event AdminFeeChanged(uint256 indexed oldFee, uint256 indexed newFee);
2
3  function changeAdminFees(uint256 newAdminFee)...{
4  ⊙    emit AdminFeeChanged(retireOldAdminFee(),
       ↪ setNewAdminFee(newAdminFee));
5  ★    emit AdminFeeChanged(setNewAdminFee(newAdminFee),
       ↪ retireOldAdminFee());
6  }
```

Listing 8.2: Example of EAS Mutant

Inheritance

The CCD (Contract Construction Deletion) operator removes the whole constructor from a contract. CCD mutations are so drastic that most of them were killed in the experiment (MS = 75%). Thus, this approach can mask more subtle problems related to individual initialization statements within the constructor. Indeed, most initialization issues from the audited projects were not related to productive mutants. Thus, we revised the implementation of CCD.

Initialization Statement Deletion (ISD): ISD improves CCD by removing each statement within constructors or initialize functions, one at a time. As a result, it prevents independent initialization operations from taking place.

```
1  constructor(...) {
2     _grantRole(ADMIN_ROLE, msg.sender);
3     baseURI = _bURI;
4     ⊙ updateRoyaltiesAddresses(_FAddress, _DAOAddress);
5     ★ //updateRoyaltiesAddresses(_FAddress, _DAOAddress);
6  }
```

Listing 8.3: Example of ISD Mutant

Function Modifiers

Mutation operators for function modifiers target both custom, user-defined modifiers, and default visibility modifiers.

Custom Modifiers

The MOD (Modifier Deletion) operator disclosed many instances where the unhappy path of a function modifier was not adequately tested. As such, it is particularly useful and intuitive for test improvement purposes. Instead, the Modifier Insertion (MOI) operator can sometime help to disclose missing *validations*, but it often generates mutants that are not very informative and only add noise. When a modifier is added

to a function signature and it survives, its analysis does not necessarily translate into useful test cases or modifications to the code under test. Additionally, more complex *access control* issues require more elaborated mutations to be disclosed, as shown by the limited generation of productive mutants in Table 8.3. For example, one common problem reported by auditors emerges with the usage of OpenZeppelin's Access Control libraries[3]. Due to an implementation oversight, a privileged role can be left vacant after revocation, potentially locking critical functionalities. The operator set was therefore enhanced as follows.

Modifier Insertion (MOI): The MOI mutants can sometimes help to reason about the contract requirements and draw attention to missing *validations* or *access control* checks. For instance, consider the IP mutant generated by MOI in Listing 8.4. This mutant targets `requestCollectionSeeds`, a function that is responsible for updating information about collections managed by the smart contract. As per the inspected audit report, any function interacting with collections should ensure the validity of the `_collectionId` parameter. In the audited project, some functions correctly implement this check through the `onlyValidCollectionId` modifier, but many functions do not. In this project, MOI applied the `onlyValidCollectionId` modifier to all functions with compatible parameters (i.e., those that take in input a `_collectionId`). Thus, the MOI mutant in Listing 8.4 effectively patched a vulnerable contract function. Still, when a modifier is added to a function signature and it survives, its analysis does not necessarily translate into useful inisghts. The MOI operator was therefore refined to reduce noise. For instance, MOI no longer targets pure or view functions in the contract.

```
1  /**
2   * Requests an update to the collection's seeds.
3   * @param _collectionId Collection ID.
4   */
5   ⊙ function requestCollectionSeeds(uint256 _collectionId) external
      ↪ onlyRole(DEFAULT_ADMIN_ROLE) {
6   ★ function requestCollectionSeeds(uint256 _collectionId) external
      ↪ onlyRole(DEFAULT_ADMIN_ROLE)
      ↪ onlyValidCollectionId(_collectionId) {
7   {
8       // ... requestCollectionSeeds logic ...
9   }
```

Listing 8.4: Inspection-Productive Mutant generated by the MOI Operator

Modifier Order Change (MOC) and Modifier Replacement (MOR): Most mutants generated by MOC during the experiments were equivalent, as they changed the orders of access control modifiers applied to the same function. Since in most cases such mutation does not affect the contract behavior, the operator was discarded. The MOR operator, which

[3]OpenZeppelin Access Control: https://docs.openzeppelin.com/contracts/2.x/access-control

substitutes one modifier for another, was excluded as well. Indeed, the effects of its mutations can be replicated through a combination of insertions (MOI) and deletions (MOD), without introducing the risk of masking effects.

Revoke Role Insertion (RRI): The RRI operator targets smart contract that use OpenZeppelin's access control libraries. First, it identifies functions that can only be invoked from a privileged account, based on the specific access control library imported by the contract. For example, the `burn` function in Listing 8.5 uses the `hasRole` method provided by `AccessControlUpgradeable` to check if the transaction sender has been granted the `ADMIN_ROLE` (line 4). When a viable function is found, RRI inserts two statements to forcefully revoke the transaction sender's privileged role (line 6) and immediately grant it back (line 7). In this way, the mutant can only be killed when the `revokeRole` operation on line 6 fails, and not because its effect are detected by unrelated tests. Mutant survival may indicate that the contract allows the only privileged user to renounce its role. It might also imply that multiple privileged accounts are set-up before testing, and the single-admin scenario is ignored.

```
1  contract Project7 is AccessControlUpgradeable {
2
3   function burn(address from, uint256 amount) public {
4      require(hasRole(ADMIN_ROLE,msg.sender));
5      _burn(from, amount);
6  ★   revokeRole(ADMIN_ROLE,msg.sender);
7      _grantRole(ADMIN_ROLE,msg.sender);
8   }
9
10 }
```

Listing 8.5: Example of RRI Mutant

Visibility Modifiers

Visibility mutations applied by FVR (Function Visibility Replacement) and VVR (Variable Visibility Replacement) did not help to improve tests or disclose issues. FVR also ranked as the second highest operator in terms of generated mutants. However, since specific replacements can indicate gas optimization opportunities, FVR and VVR were included in a cluster that auditors can independently enable to focus on code optimizations.

Operators

Mutations involving operators were positively received by auditors, with few exceptions.

Assignment Operator Replacement (AOR): Compound assignment replacements can reveal instances of values not being checked by the tests after an update, and help to disclose *Miscalculations*. However, multiple replacements on the same target rarely yield extra insights. The operator was refined to replace each compound assignment solely with the

```
1 function foo(uint a,uint b) public returns(uint){
2 ⊙     return a / 100 * b;
3 ★     return a *b / 100;
4 }
```

Listing 8.6: Example of PLR Mutant

simple assignment, the most surviving replacement across all projects.

Binary Operator Replacement (BOR): Relational operators are highly prevalent and linked to many TP mutants. To mitigate their testing time overhead, we pinpointed specific replacement combinations that support the analysis of positive and negative test scenarios and boost edge case coverage. Mathematical operators replacements, while not as frequent, often yielded redundant outcomes. These were similarly optimized considering the most subtle increment and decrement replacements.

Increments Mirror (ICM): The ICM operator replaces compound assignments with their mirrors (e.g., $-= \rightarrow =-$). Ultimately, ICM was removed from the set due to the limited insights it provided and its overlap with the AOR mutants.

Precision Loss Replacement (PLR): The limited precision of Solidity's fixed-point system can lead to inaccuracies when a division precedes a multiplication. As shown in Table 8.3, simply replacing mathematical operators (BOR) or numerical values (ILR) is not the best approach to disclose *precision losses*. The new PLR operator swaps the order of multiplication and division in binary or tuple expressions to simulate a precision loss. As shown in Listing 8.6, PLR also targets cases where division precedes multiplication, prompting developers to evaluate whether the original order is intentional or indicative of a potential bug.

Types

Mutation operators for Solidity types are categorized into data types, data location and conversions.

Data Types

The AVR (Address Value Replacement) operator swaps address values with special address function calls (e.g., `address(this)`) or with a different hardcoded address declared in the contract. The effects of an address mutation can vary largely and help thoroughly validate the effectiveness of a test suite. However, replacing each target with both `address(0)` and `address(this)` often resulted in redundant mutations. In all evaluated projects, if the first replacement survived or was killed by the tests, the second replacement yielded the same outcome. Moreover, the address faults injected by AVR should be replaced with more contextually relevant mutations, as hinted by its rather high Mutation Score. Lastly, AVR overlooked several mutation opportunities

associated with non-literal address assignments. For example, one high-severity *access control* finding from Project 4, shown in Listing 8.7, reveals that any non-whitelist account (line 2) can delegate voting power to a whitelisted address. In this scenario, replacing the address assigned to the delegator with a different one could have helped to expose the problem. The numeric and boolean value-targeting operators - Integer Literal Replacement (ILR) and Boolean Literal Replacement (BLR) - generated a substantial number of TP and IP mutants for the issues we analyzed (See Table 8.4), most of which were related to *Miscalculation*, *Precision Loss* and *Validation*. However, similar to AVR, their rules were restricted to literal values, missing out on various mutation opportunities.

```
1 function delegate(address delegatee, uint256 id) public virtual
    ↪ override {
2     address account = _msgSender();
3     _delegation[id][account] = delegatee; ...
4 }
```

Listing 8.7: Issue related to whitelisted address from Project 4

Address Value Replacement (AVR): Drawing from previous insights, we refined AVR to incorporate mutations based on contextual data. AVR can now identify additional mutation opportunities by preemptively gathering address identifiers within the contract. For example, it can replace the right-hand side of an address assignment with a function parameter, a state variable identifier, or another variable identifier declared in the same function. In cases where multiple identifiers are available, AVR prioritizes them based on their proximity to the target, thereby enhancing the probability of injecting a realistic bug. If no other suitable identifier is available, only then is the target replaced with the standard zero address.

Number Value Replacement (NVR), Boolean Value Replacement (BVR) and String Value Replacement (SVR): The NVR, BVR and SVR operators include enhanced rules capable of mutating the right-hand side of simple integer, boolean and string assignments or declarations, even when the target is not a literal. This is achieved by first gathering the name and type of each function argument, local variable and state variable. Array mutations can assist auditors in verifying the correctness of content updates. While valuable index access mutations were generated, SuMo lacked operators for targeting array manipulations through dedicated member functions.

Array Member Replacement (AMR): The AMR operator was introduced to ensure the correct state of arrays after an update operation. AMR comments out any `pop()` and `push()` invocation to simulate a missing element deletion and insertion. It also removes the argument of a `push()` to simulate the insertion of the wrong element into the array.

Data Location

The Data Location Replacement (DLR) operator, which swaps the `memory` and `storage` data location keywords, was not deemed useful for improving tests or disclosing issues. However, a promising research direction emerges when considering its inclusion in a cluster of operators tailored for optimizing gas consumption. For example, a live `storage` to `memory` mutant suggests that storage might be unnecessary for the data in question, as state changes are not involved.

Conversions

The ECS (Explicit Conversion to Smaller Type) operator replaces each `uint` and `bytes` casting with its smallest available type (e.g., replacing `uint256` with `uint8`). However, these replacements can be too noticeable in tests because of their significant impact. SuMo also missed several mutation opportunities due to its lack of support for SafeCast, a popular OpenZeppelin's library[4] that adds overflow checks to Solidity's casting operators. Lastly, we observed that simple ECS mutations fall short in uncovering complex casting-related issues such as aliasing (see Table 8.3).

Casting Expression Replacement (CER): The new CER operator is a more subtle version of ECS that only decreases type castings by one step. Moreover, it replaces each SafeCast function call with a regular cast. Only tests that expect a revert upon overflow can detect this type of mutant, forcing the developers to verify such scenario. Instead, if the contract imports the SafeCast library and yet a regular casting expression is found within its code, CER injects the opposite mutation. In this case, a live mutant likely indicates a patch opportunity.

Address Aliasing Check (AAC): The new AAC operator can expose *aliasing* issues by inserting multiple statements before a downcasting operation. Listing 8.8 shows an aliasing issue found by auditors in one of the considered projects. Here, `_getAddressFromId(bytes32 id)` downcasts 32 bytes to an address, discarding the lower 12 bytes. As a result the same address can be returned for different values of `_id`. AAC first converts the target identifier to a dynamic byte array, then it applies the logical NOT operator to its last byte. By returning the address downcasted from the mutated identifier, it is possible to disclose the problem.

Function Parameters and Return Variables

The RSD (Return Statement Deletion) operator revealed instances where the return value of a function was not tested. Live RSD mutants allow auditors to evaluate the necessity of the return value and consider potential code refactoring for improved clarity. However, it is crucial to use RSD in conjunction with other operators, as certain functions can affect the contract state without returning any value.

[4]SafeCast: `https://docs.openzeppelin.com/contracts/5.x/api/utils#SafeCast`

```
function _getAddressFromId(bytes32 _id) external returns (address) {

⊙     bytes memory arr = abi.encodePacked(_id);
⊙     arr[arr.length-1] = @~@arr[arr.length-1];
⊙     return address(bytes20(arr));

★     return address(bytes20(_id));

}
```

Listing 8.8: Example of AAC Mutant

```
function withdraw(...) public virtual override returns (uint256) { ...
⊙ return super.withdraw(assets, receiver, owner);
★ return super.withdraw(assets, receiver, owner);
}
```

Listing 8.9: Example of improved RSD Mutant

Return Statement Deletion (RSD): Deleting a statement that returns the outcome of a state-changing function is a sub-optimal mutation. This mutant could be killed by unrelated tests that verify the effects of the function call on the contract state, even if the return statement itself is never checked. To avoid this masking effect, RSD now removes the `return` while retaining the original function call (see Listing 8.9).

Special Variables and Functions

Mutation operators for special variables and functions involve block and transaction properties, error handling and members of address types.

Block and Transaction Properties

Operators targeting block and transaction properties - GVR (Global Variable Replacement) and TOR (Transaction Origin Replacement) - can hold significant value for auditors. As noted in Table 8.4, GVR produced a TP mutant related to a time miscalculation issue, and an IP mutant with the potential to reveal a high-severity *block variable dependence*. The latter simulated the actions of block proposers manipulating variables to achieve a desired outcome, influencing the behavior of the smart contract. However, replacing each block variable with every compatible option resulted in many redundant mutations, inflating the Mutation Score and wasting time.

Global Variable Replacement (GVR): The GVR operator was revised to replace block variables that are normally used as a source of randomness (e.g., `block.timestamp`) with `block.prevrandao`. The survival of this mutant likely points to a dangerous dependency where `prevrandao` should be used instead. Indeed, such a drastic

mutation is more likely to be killed if the target variable is used for other purposes (e.g., to trigger time-dependent operations). Other integer block and transaction variables, which can affect gas, Ether, or time-related computations and checks, are incremented and decremented by one. Lastly, `blockhash(uint blockNumber)` is replaced with 0. Indeed, this function always returns 0 if the input block number is older than 256 blocks. Since this known behavior could be exploited it is important to carefully evaluate its usage within the contract.

Transaction Origin Replacement (TOR): The TOR operator helps to identify risks associated with using `tx.origin` for authentication by swapping it with `msg.sender` (and vice versa). In our experiments, all 119 mutations replaced `msg.sender` with `tx.origin` (due to the less frequent usage of the latter). However, the effort required to kill such mutants may outweigh the benefits. Indeed, tests that simulate transaction chains are impractical to define and not typically included in test suites. We therefore removed this inefficient rule from TOR.

Error Handling

Mutations injected by the EHC (Exception Handling Change) operator were well-received by auditors when they remove entire error handling statements (e.g., `require`) from the contract. Indeed, the low Mutation Score achieved by EHC shows how test suites submitted for audits inconsistently explored error scenarios. Despite only generating TP mutants in the experiments, EHC may also uncover interesting fix opportunities during code inspection, such as the usage of redundant error checking.

Exception Handling Deletion (EHD): We changed EHC into EHD to only inject statement deletion mutations. Additionally, we revised EHD to include support for custom Solidity errors, allowing the evaluation of the test suite's effectiveness in handling user-defined error expressions.

In the context of error handling, we noticed a common issue across projects: *incorrect* or *missing validations*. These generally indicate that a function is lacking proper input value checks, or that the contract state is not being verified before performing a certain action. Since mutation testing does not rely on program specification, missing validations are inherently difficult to reveal. SuMo exposed some of them through different combinations of mutants, but a dedicated operator would provide auditors with more immediate feedback.

Require Statement Insertion (RSI): The RSI operator was designed to reason about *missing validations*. While it may not be possible to identify the location where a specific check is missing, the repetitiveness of validation statements in smart contracts offers an opportunity to target common oversights. RSI analyzes all the require statements in a contract and determines whether they should be injected into other functions. Compatibility is based on the target function parameters and the existing require statements in its body. For example, the `mint` function in Listing 8.10 lacks a check to ensure the token recipient `to` is unfrozen. This validation is however present in the `transfer` function of the same

contract. RSI can copy the `require` in the mint function, forcing the auditor to evaluate its survival.

```
1 function mint(address to,uint256 amount) public {
2 ⋆ require(!_frozen[to]);
3     // mint logic ...
4 }
5
6 function transfer(address to,uint256 value) ... {
7     require(!_frozen[to]);
8     // transfer logic ...
9 }
```

Listing 8.10: Missing Input Validation from Project 7 introduced by RSI Mutant

Members Of Address Types

A common issue in audited projects involves *unchecked return values* from Ether and token transfers. The ETR (Ether Transfer Replacement) operator implemented by SuMo helped to disclose one such case (see Table 8.4) by swapping `transfer()` and `send()`, but the other injected replacements did not provide useful feedback. Most importantly, its mutation rules did not differentiate between transfers of Ether and tokens, generating uncompilable mutants in the latter case.

Transfer Return Check (TRC): The TRC operator overcomes the limitations of ETR by injecting different mutations based on the type and context of the function call:

a) Reverting Function Calls: It replaces `transfer` with `send` to return false on failure, highlighting missing test coverage for reverting transfers.

b) Standalone Non-Reverting Function Calls: It encloses with a `require` statement those standalone Ether or token transfers that return false on failure. The survival of this mutant can reveal a patch opportunity and encourage the derivation of tests for the failing transfer scenario.

c) Nested Non-Reverting Function Calls: It mutates any nested, non-reverting function call differently depending on its context (see Listing 8.11). If the success of the transfer is a prerequisite for the execution of some logic, the surrounding statement (e.g., a `require`) is removed. Mutant survival implies that the tests did not adequately cover the failing transfer scenario (e.g., due to insufficient funds). Instead, removing the whole require statement prevents the transfer operation from being executed. In this case, a live mutant reveals that the transfer success and the relative state changes, such as account balances consistency, were not verified. If a variable or a return statement are used to propagate the return value of a transfer, TRC unconditionally returns `true`, with and without executing the transfer operation.

Standard Functions

SuMo includes several operators for mutating global function calls, regular function

```
1        // Require statement
2  ⊙     require(super.transfer(to,amount));
3  ★     ~~require(~~super.transfer(to,amount)~~);~~
4
5        // Return statement
6  ⊙     return super.transfer(to,amount);
7  ★     ~~return~~ super.transfer(to,amount); ~~return true;~~
8
9        // Variable declaration
10 ⊙     bool success = super.transfer(to,amount);
11 ★     bool success = ~~true~~;
```

Listing 8.11: Example of TRC Mutants for Nested Transfer Calls

definitions and return statements. However, there are no operators for verifying the effects of standard function calls on the smart contract state.

Function Call Deletion (FCD): The FCD operator deletes those function calls that are not embedded into other expressions (e.g., assignments, conditional statements, or other function calls). Surviving FCD mutants help to validate the contract state after a function call, while offering valuable insights into its practical impact and necessity.

Variable Units

The VUR operator targets Solidity's ether and time units. Although VUR could have helped to disclose a *gas usage* issue, mutations involving Ether units can be overly drastic (e.g., converting `ether` to `wei`), causing compilation errors. Moreover, swapping each time unit with all other possible units generates redundancy.

Variable Unit Replacement (VUR): VUR now replaces each variable unit with its neighbor, reducing the likelihood of compilation errors or out-of-gas exceptions.

8.5.2 Discussion of Results

The analysis of the results already presented in Section 8.4 already suggests that SuMo can enhance auditing capabilities. However, we also identified weak spots and opportunities for improvements that would make SuMo more actionable for the code review process. In this Section, we assess the effectiveness of the revised mutation operators. Table 8.5 shows the results of the retrospective study we conducted on the selected projects using the new mutation strategy.

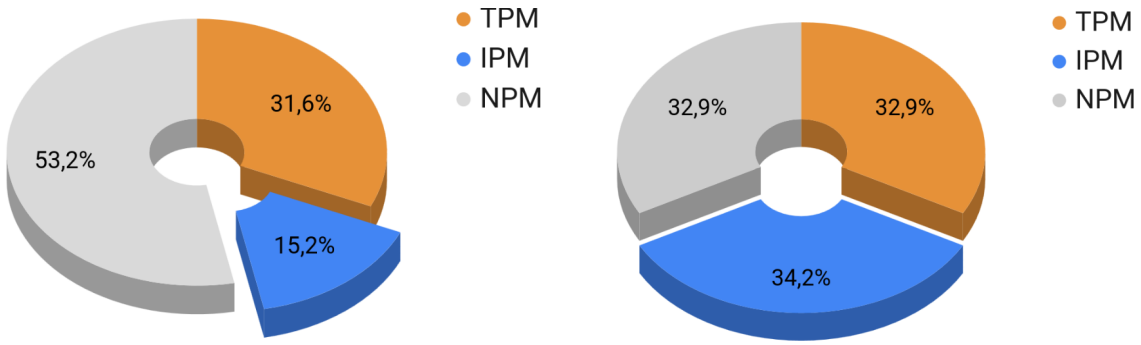
Although the revision of the operators introduced new mutation rules, optimizations led to a 30% reduction in the total number of generated mutants, resulting in time savings of over 114 hours. Meanwhile, the ratio of valid mutants generated by SuMo increased from 84.8% to 92.5%, showcasing a more effective generation process.

Table 8.5: Analysis of Productive Mutants Generated by the Enhanced Operators

ID	#M	#V	MS	Total MTRI	With NPM	With TPM	With IPM	With both TPM & IPM
1	2578	2341	54.2	6	3	3	0	0
2	425	410	66.4	12	6	5	0	1
3	2411	2218	61.3	9	6	2	1	0
4	1428	1336	46.9	22	5	10	4	3
5	1918	1779	61.1	8	2	2	3	1
6	730	635	77.3	6	2	2	2	0
7	125	125	71.2	3	1	0	1	1
8	3491	3275	48.9	13	1	2	3	7
Tot.	13108	12121	-	79	26	26	14	13

Legend:
M: Generated Mutants, V: Valid Mutants, MS: Mutation Score, MTRI: Mutation Testing Relevant Issues, NPM: No Productive Mutants, TPM: Test-Productive Mutants, IPM: Inspection-Productive Mutants

Generation of Productive Mutants Figure 8.1 shows that new set of mutation operators generated productive mutants for 67% of the Mutation Testing Relevant Issues (MTRI), marking a significant improvement over the original set's 47%. Of these issues, 32.9% were attributed to Test-Productive mutants, while 34.2% were related to Inspection-Productive mutants. Notably, the number of issues related to IPM more than doubled following the enhancement of the mutation operators.



(a) Original SuMo operators (b) Enhanced SuMo operators

Figure 8.1: Issues linked to Productive Mutants

Categories of Issues Linked to Productive Mutants Table 8.6 shows how the new mutation strategy increased the generation of productive mutants for specific issue types. For each category, it reports the number of MTRI associated with NPM, TPM, and IPM, along with their respective deltas compared to the original experiment (refer to Table 8.3). While the original SuMo operators rarely helped to disclose *unchecked transfers*,

with the introduction of the TRC operator all of them could be identified through mutant inspection. A similar scenario can be observed for the *access control* category. The RRI operator played a crucial role in the disclosure of revocable ownership and roles, while the enhanced AVR and NVR operators generated useful mutants for project-specific issues (e.g., see Listing 8.7). For the *precision loss* category, in the previous experiment two issues were linked to NPM and TPM mutants, respectively. Now, both can be immediately detected through code inspection. Instead, no productive mutants were associated with the *reentrancy* category. This was expected, as even the most elaborate mutations may not effectively target the specific conditions and patterns that underlie such vulnerabilities. Mutation testing also proved generally unsuitable for detecting *missing initialization* and *validation checks*. These issues are characterized by the omission of code, rather than a specific code segment that can be modified through mutation. While certain insertion operators, such as MOI and the novel RSI, did help uncover validation issues, their effectiveness remains context-dependent.

Table 8.6: Category of Issues with associated Productive Mutants for the Enhanced Operators

Category of Issue	Total MTRI	With NPM	With TPM	With IPM	With both TPM & IPM
Access Control	7	1 (−5)	0	4 (+3)	2 (+2)
Aliasing	1	0 (−1)	0	1 (+1)	0
Block Variable Dependence	1	0	0	1	0
Checks−Effects−Interactions Violation	2	0 (−1)	2 (+1)	0	0
Gas usage	4	0	3	0	1
Integer Over/Underflow	1	0 (−1)	1 (+1)	0	0
Initialization	3	2	1	0	0
Incorrect Event Log Usage	1	0	0	1	0
Miscalculation	11	2 (−1)	7 (+1)	0	2
Logical Oversight	4	1	1 (−1)	0	2 (+1)
Other	3	2	1	0	0
Precision Loss	4	0 (−1)	0 (−1)	2 (+1)	2 (+1)
Reentrancy	3	3	0	0	0
Timing	1	0	1	0	0
Unchecked Transfer	5	0 (−4)	0	5 (+4)	0
Validation	28	15 (−2)	9	0	4 (+2)
Total	79	26 (−16)	26 (+1)	14 (+9)	13 (+6)

Overall, while it remains crucial to complement SuMo with static analyzers and other specialized tools, the value of investing in mutation analysis should not be underestimated. The process of test improvement inherently involves examining mutants, which can also reveal flaws in the code under test and support contract patching and refactoring.

Key Insights: *Enhancement of the Mutation Operators*

Feedback from auditors is fundamental to derive heuristics that boost the generation of productive mutants. The enhanced operators could have helped to discover about 67% of the analyzed issues. Half of these were linked to Inspection-productive mutants that do not require the derivation of new test cases.

8.6 Threats to Validity

This section outlines the potential threats to the validity of our study and the steps taken to mitigate them.

Threats to Internal Validity A primary threat to internal validity arises from the manual nature of our analysis. Identifying productive mutants requires careful domain-specific knowledge for each project, as well as an understanding of the issues documented by auditors. Since the assessment of productive mutants is inherently qualitative, subjectivity may influence our classifications. What appears productive to one reviewer may not be perceived as useful by another.To reduce bias, we sought feedback from auditors about the impact and characteristics of the inspected issues.

Threats to External Validity The generalizability of our findings is constrained by the size and scope of the experimental dataset. Due to the manual costs associated with the manual analysis of the results, our study was conducted on a limited set of Solidity projects. We attempted to diversify the dataset by selecting projects from different categories (e.g., NFT marketplaces and DeFi protocols).

Threats to Construct Validity A threat to construct validity is the qualitative and exploratory nature of our interviews with auditors. Our auditor feedback was collected through informal, one-on-one interviews with three participants, which, while insightful, represents a small sample size. Since auditors may have different perspectives and experiences, a larger and more systematic study would be required to establish more broadly applicable heuristics for identifying productive mutants. To fully capture auditor-driven heuristics, we would need a more structured mechanism that automatically collects auditor feedback on mutants during code review (similar to how Google incorporates developer feedback into their mutation analysis pipeline [139]).

8.7 Related Work

The usage of mutation testing to enhance smart contract test suites has been extensively explored in prior studies [3, 22, 40, 75, 76, 100]. However, the relationship between mutants and code inspection activities remains largely unexplored in the blockchain domain. This Section reports related efforts in integrating mutation testing in industrial settings, and investigations on the productiviy of mutants.

Industrial Integration of Mutation Testing Mutation testing has seen growing adoption in industrial settings, with companies like Google and Facebook integrating it into their software development workflows. Google, for instance, has developed a scalable approach to mutation testing that operates on changed lines of code during code reviews rather than applying mutation analysis to the entire codebase [137]. This strategy reduces computational costs while ensuring that reported mutants remain relevant to developers. Additionally, Google employs mutant suppression heuristics based on developer feedback to filter out unproductive mutants. A key component of this integration is Critique, an internal code review system which allows reviewers to endorse a mutant using a "Please fix" button. This feedback mechanism helps refine which mutants are reported over time and ensures that mutation testing remains a valuable part of Google's code review process. Facebook's approach, known as Mutation Monkey [26], takes a different direction by leveraging machine learning to learn error-inducing patterns from historical bug fixes and operational anomalies. Instead of generating many mutants, Facebook focuses on creating a small, highly actionable subset of mutants that have a higher likelihood of exposing untested code. These mutants are surfaced to developers during code review, along with coverage information indicating which tests visited but failed to detect the mutation.

Characteristics of Productive Mutants A key challenge in mutation testing is identifying and surfacing mutants that provide meaningful insights to developers. Prior research suggests that productive mutants are those that either (1) elicit new and effective tests or (2) highlight issues or ambiguities in the code, leading to knowledge improvements [139]. Using context-aware selection based on past mutant performance, Google has improved the productivity rates of reported mutants, with developers marking 82% of surfaced mutants as actionable. A recent study within Google [138] analyzed over 1500 mutant-related merge requests, shedding light on their value in code inspection and review. While test improvement emerged as the most common approach for resolving mutants, they also observed instances of code refactoring and in-depth discussions about the codebase. Facebook's Mutation Monkey approach [26] aligns with this principle by minimizing the number of reported mutants and ensuring that each mutant has a high survival rate. In an empirical study with 26 developers, almost half of the participants actively modified or created new tests in response to surfaced mutants.

8.8 Directions for Future Research

This study assessed the practical usefulness of mutation testing for smart contract auditing by analyzing a set of Solidity projects submitted to Quantstamp for review. Our findings highlight the value of mutation analysis in the auditing process, demonstrating that live mutants can aid in both test improvement and manual review by surfacing flaws in the code under test. However, despite these benefits, a significant challenge remains in effectively filtering and presenting productive mutants to auditors, as the sheer number of generated mutants can create cognitive overhead.

To address this, future research should focus on developing automated mechanisms to prioritize productive mutants and filter out uninformative ones, reducing noise and mak-

ing the review process more efficient. Leveraging machine learning models or heuristic-based approaches could allow mutation testing tools to rank mutants based on their likelihood of exposing issues. To this end, it is essential to enable the real-time collection of feedback by integrating a system where auditors can classify mutants as useful or irrelevant during the review process, similar to infrastructures seen in developer-driven mutation testing at Google [135].

Beyond tool-level enhancements, an important avenue for future empirical work is the systematic study of mutant prioritization strategies in real-world auditing contexts. One potential research design could involve deploying ranked sets of live mutants, prioritized by different heuristics, across multiple audit sessions. It would then be possible to collect both quantitative data (e.g., time spent per mutant) and qualitative feedback (e.g., perceived utility) from professional auditors. This would help identify which prioritization criteria are most effective in practice.

Overall, addressing the prioritization problem is not only a technical challenge but also a socio-technical one. A more refined understanding of how auditors interact with mutants in realistic workflows will be essential for integrating mutation testing in the smart contract domain.

Acknowledgements

This research has been supported by Quantstamp.

CHAPTER 9

MUTATION TESTING OF SMART CONTRACTS AS A SERVICE

Smart contracts are self-executing programs that run on a blockchain and, due to their complex and specialized nature, often require audits by independent parties prior to deployment. A promising approach for enhancing their reliability is mutation testing, a powerful yet time-intensive technique for assessing test adequacy. This Chapter presents a framework designed to streamline and parallelize the mutation testing process, making it more practical and accessible for auditors. Simulations conducted on real-world Solidity projects demonstrate that the proposed service substantially reduces the computational time required by the stand-alone SuMo module, delivering results to auditors more quickly.

This chapter is based on material from the following publication: [23]

* * *

9.1 Introduction

Blockchain is a transformative technology that has revolutionized transaction processes by offering high levels of security and transparency. At the heart of this transformation are smart contracts, which are self-executing agreements running on the blockchain. The immutable nature of these contracts calls for rigorous testing before deployment, since vulnerabilities can lead to significant financial losses and undermine trust in the underlying implementation [208, 187, 20]. To support the development of secure blockchain-based applications, the literature has introduced a variety of techniques, tools, and approaches [185]. Nonetheless, the specialized nature and the complexity of smart contract security has prompted developers to ask for independent secuirty reviews from third-party auditors [130, 39]. The **auditing** process is crucial for identifying and mitigating hidden vulnerabilities, errors, and potential exploits that could compromise distributed applications [2].

One particularly valuable technique for auditors is **mutation testing** [10], which assesses a software test suite by examining whether it can detect small mutations introduced into the source code. Despite its proven capacity to uncover bugs and foster deeper code understanding [10, 139], mutation testing has been slow to gain traction in the smart contract domain for several reasons. As discussed in Section 4.3.1, existing mutation testing tools for Solidity projects lack standardization, making their setup and operation complex. Auditors consequently have to rely on different mutation operators for different projects, preventing the establishment of a common benchmark for test suite quality. In addition, running mutation testing is notoriously time-consuming, so returning results to auditors in a timely manner can be challenging. Because of this, mutation testing is frequently perceived as a high-cost activity with limited immediate benefits. For broader adoption, tools must offer infrastructure upgrades that facilitate cross-project usage and enhance usability, for instance by providing intuitive interfaces for visualizing and interpreting results. Most importantly, results must be delivered early in the audit cycle, allowing sufficient time for analysis.

To address these issues, we propose an end-to-end framework that **streamlines** and **sppeds-up** mutation testing for smart contracts. This framework builds on SuMo, which already supports Solidity projects that employ different testing frameworks [22]. Although SuMo is effective as a stand-alone module, running it locally can be resource-intensive and slow when dealing with large projects. Our solution introduces a remote service for distributing mutation testing tasks across parallel Docker containers, reducing the required time and computational resources. We also present infrastructure enhancements, including result visualization features that improve usability and facilitate the integration of this technique into standard auditing workflows. In our empirical evaluation, we compare five distinct approaches for distributing work across three open-source Solidity projects, providing guidelines on selecting an optimal distribution strategy based on project specifics and resource constraints. The simulations indicate that the most effective strategy can reduce local, sequential mutation testing time by an average of 70%.

The rest of this chapter is structured as follows: Section 9.2 presents the research objectives, providing a deeper understanding of the context in which SuMo operates. The framework we propose is detailed in Section 9.3, while Section 9.4 discusses its validation through simulations on real-world projects. Section 9.5 presents related work in the field of blockchain and mutation testing. Lastly Section 9.6 summarizes our findings and explores directions for future research.

9.2 Research Objectives

This thesis aims to make mutation testing a practical and valuable addition to the smart contract auditing toolkit. The identified objectives are as follows.

1) Streamlining Mutation Analysis A primary objective is to provide a cohesive framework that enables auditors to run mutation tests on Solidity projects with minimal complexity. This involves simplifying the process of initiating mutation testing, retrieving outcomes, and interpreting mutant behaviors in a practical manner. To achieve this,

we have extended the stand-alone SuMo tool with two core additions: a Visual Studio Code extension and a service deployable on a remote node. The VSCode extension allows auditors to launch mutation testing tasks and inspect results within the same development environment, improving the user experience. The remote service tracks these mutation testing jobs across different projects and preserves the Mutation Score over successive revisions. This feature also helps auditors evaluate the effectiveness of newly introduced tests in case of a follow-up audit.

2) Reducing Time Costs Another key objective is to mitigate the time costs of mutation testing. This is fundamental for two reasons. First, auditing is a thorough, complex process that can delay the release of a project. Mutation testing, by its nature, is a time consuming technique that can further extend the auditing time-frame. Thus, if auditors plan to analyze mutants and provide test improvement tips to clients, the results should be made available to them as quickly as possible. Second, our previous study [10] in collaboration with *Quantstamp* showed that mutation testing can play an important role during manual code inspection. Analyzing live mutants can not only lead to test improvements, but also increase code knowledge and disclose issues in the contracts under test. However, to make the most of what we refer to as Inspection Productive mutants, results should be presented to the auditors before the code review phase is complete. By providing SuMo as a service, we allow auditors to run parallel mutation testing tasks in the cloud, offering scalability and reducing execution time. This approach can speed-up the overall process and ensure that the insights gained from live mutants are available at a point when they can still meaningfully influence the audit.

9.3 SuMo As A Service

In the design of our mutation testing framework, we recognized the need for a solution that offered efficiency, scalability and ease-of-use for auditors. One effective technique to speed-up mutation testing is parallel mutant execution [141]. State-of-the-art tools like PiT [140] typically support local parallelization to improve the efficiency of the testing process. However, a local approach is limited by the computational power and memory of a single machine. Additionally, it can introduce reliability issues due to the lack of isolation between parallel testing tasks, potentially leading to failures and inconsistent results. This is an important consideration for Solidity projects, where testing contracts typically involves setting up a test network that simulates the blockchain environment. To address the limitations of local parallelization, several works explored distributing mutation testing executions across multiple nodes [110, 37], such as Amazon EC2 instances [114]. While cloud services like EC2 offer scalable access to computational resources, they also incur higher costs based on time and resource consumption. Moreover, they require careful planning to ensure data security and compliance, especially when handling sensitive codebases.

In this thesis, we propose a framework that leverages Docker containers to parallelize mutant execution tasks. This approach offers portability, robustness, and good scalability. Indeed, each container operates in isolation, ensuring that testing environments do not interfere with each other. Docker containers can be created and shut down in seconds,

which is ideal for mutation testing where many relatively short-lived environments must be spawned to run testing tasks. Furthermore, mutation testing can be run on any node equipped with Docker, scaling the number of parallel containers based on the workload and available infrastructure. The proposed framework consists of three main components:

1. The **SuMo Module** retains the core functionality described in our previous work [22], requiring the user to install it in a Solidity project via npm.

2. The **SuMo Service** can be deployed on a host to handle remote, parallel mutation testing jobs.

3. The **SuMo Extension** is a companion Visual Studio Code extension for SuMo.

Since the SuMo module was comprehensively described in Section 6, in the following we only present the SuMo Service and Extension.

9.3.1 The SuMo Service

The SuMo Service is a complementary solution that can be deployed on a host to handle remote, parallel mutation testing jobs. This offers auditors the flexibility to run mutation testing either locally or in the cloud. The service is implemented as a Spring application that leverages Java's robust support for multithreading to ensure scalability and efficient handling of concurrent requests. As introduced above, clients can interact with the service through the SuMo VSCode extension. In particular, they can upload the Solidity Project Under Test (PUT) and the relative SuMo configuration to the service, which will start the testing process. Specifically, we define:

1. A **Mutation Testing Job**: the execution of a **complete** mutation testing run on a PUT. A mutation testing job can be split into parallel mutation testing tasks executed within individual Docker containers;

2. A **Mutation Testing Task**: the execution of a **partial** mutation testing run on a PUT, wherein only a subset of the generated mutants are tested within a Docker container.

The SuMo service can run parallel mutation testing tasks on the same PUT, and concurrent jobs over different PUTs. This logic is handled by two core components (See Figure 9.1):

1. The **Mutation Service** receives the PUT and prepares it for a mutation testing job. In particular, it defines and allocates parallel execution tasks through the Docker Service.

2. The **Docker Service** starts and manages the execution of mutation testing tasks within parallel Docker containers run on the same host. Each container hosts a copy of the PUT and its dependencies, including SuMo.

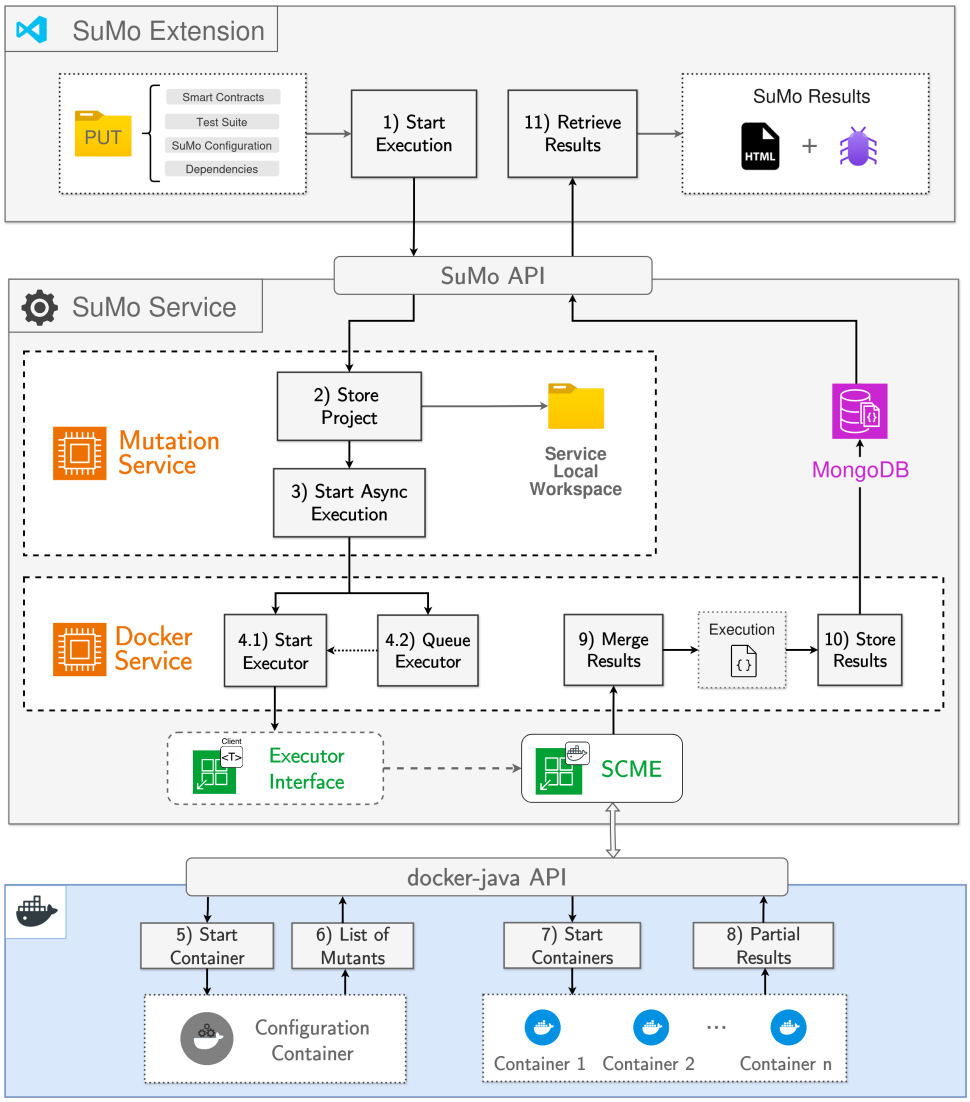


Figure 9.1: SuMo Service Architecture and Workflow

The *Docker Service* manages the workload distribution and the execution of containers by creating an **Executor** in charge of running SuMo on the PUT. Docker executors are a concept commonly associated with CI/CD pipelines, where they refer to the agents or workers responsible for running individual pipeline jobs within containers. Similarly, we utilize Docker executors for managing mutation testing tasks and jobs. The SuMo service currently provides five custom executors, although only one can be utilized at a time: the Operator Executor (OE), Smart Combined Operator Executor (SCOE), Smart Combined Mutants Executor (SCME), Dockerized Combined Operators Executor (DCOE), and Dockerized Combined Mutants Executor (DCME). All the executors can create, start, stop, and manage containers through the Docker Client. However, the defined *Executor interface* allows the SuMo service to be integrated with different containerization technologies.

Mutation Testing Workflow Here we describe the step-by-step workflow of a remote mutation testing job executed through the SuMo service, as depicted in Figure 9.1. The

process begins when a user uploads the Solidity Project Under Test (PUT) from their VSCode workspace to the SuMo service ①. When the *Mutation Service* receives the PUT, it unpacks it in a workspace directory ②. Then, it activates the *Docker Service* ③, which is responsible for creating the specific executor chosen by the user. The *Docker Service* checks if the host's available resources are sufficient to run SuMo on the PUT. If resources are insufficient the task is queued ④.2, otherwise mutation testing can start ④.1. For instance, in the workflow depicted in Figure 9.1, the selected executor is the *Smart Combined Mutant Executor* (SCME). This executor first creates a *Configuration Container* ⑤ where preliminary operations are performed. These operations include installing the PUT's dependencies, enabling the user-selected mutation operators, and computing the list of mutants generated by SuMo for the PUT. This list is then returned to the executor ⑥, which evenly distributes the mutants across a user-selected number of containers n. At this point, the executor can start n mutation testing tasks by running each container in parallel ⑦. Upon completion of a mutation testing task, the respective container sends its partial results back to the executor ⑧. The executor then relays these results to the *Docker Service*, which merges them into a single *Execution* object ⑨. This object is subsequently stored in the database ⑩. Once all the containers have terminated, the user can retrieve, save and visualize ⑪ the complete mutation testing results locally, using the built-in feature of the SuMo VSCode extension.

Mutation Testing Executors The Executor package of the SuMo service hosts the *Docker Client* responsible for building images, starting-up and tearing-down containers.This component currently includes five custom executors for running mutation testing tasks on a given project, each employing a distinct logic for container creation and workload distribution. Considering that mutation testing is not conducted over multiple nodes with heterogeneous hardware, we employ static distribution algorithms such as the one originally proposed by Offut et al. [125]. Such algorithms minimize overhead by splitting the workload (i.e., the mutants to be tested) only once, before starting the mutation testing process [37]. The workload is then distributed so that each container receives approximately the same quantity of mutants. The implemented executors are:

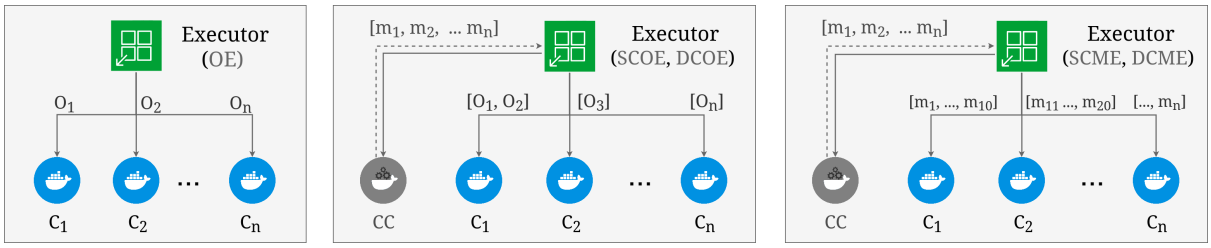


Figure 9.2: Workload distribution strategies for each Executor

OE - *Operators Executor:* The OE implements a baseline parallelization solution that assigns a single mutation operator to each container. As illustrated in Figure 9.2, OE creates a preset number n of parallel containers $C_1, ..., C_n$ from a shared Docker image, so that each container holds a copy of the PUT. When a given container C_i is started, it installs all necessary PUT dependencies and enables a single SuMo mutation operator O_i. Once all the mutants generated by O_i have been tested, C_i is terminated and a new

container is spawned in its place. This process continues until all the mutation operators enabled by the user have been run on the PUT.

SCOE - *Smart Combined Operators Executor:* The SCOE differs from OE in that it first computes the list of mutants $m_1, ..., m_n$ generated by SuMo for the PUT through a *Configuration Container CC*. SCOE then distributes the mutation operators $O_1, ..., O_n$ over a fixed number of containers $C_1, ..., C_n$, commensurate with the amount of mutants they generated. Although this does not guarantee that each container receives the exact same number of mutants, it approximates an equitable distribution by aligning the output of each operator.

SCME - *Smart Combined Mutants Executor:* The SCME works on analogous principles as the SCOE. However, instead of creating execution tasks by distributing operators, it directly allocates the generated mutants $m_1, ..., m_n$ to the spawned containers. This approach optimizes workload distribution, as the quantity of mutants per container remains nearly uniform.

DCOE - *Dockerized Combined Operators Executor:* The DCOE distributes mutation operators over a preset number of parallel containers just like SCOE. However, it uses a custom Dockerfile to build a Docker image tailored to the PUT's requirements. This approach speeds-up initial setup operations because each newly spawned container already includes all dependencies needed for running SuMo on the PUT. Furthermore, Docker caches the image after the first build, speeding up subsequent mutation testing jobs on the same project.

DCME - *Dockerized Combined Mutants Executor:* The DCME distributes the mutants evenly over a fixed number of containers like SCME, while employing the same custom image creation strategy as DCOE.

9.3.2 SuMo Extension

To provide a cohesive user experience, we implemented a companion Visual Studio Code extension for SuMo. This extension simplifies the configuration of mutation testing for projects and the inspection of live mutants. In particular, it allows auditors to easily adjust their testing preferences (e.g.: the mutation operators to be used or the contracts to be blacklisted) and start a remote - or a local - mutation testing run. As illustrated in Figure 9.3, the extension features a dedicated panel for connecting to the SuMo service hosted at a specific address and port. Once mutation testing starts, it is possible to monitor the execution progress by selecting a preferred level of log granularity, from simple error logs to detailed container-level information. Thus, auditors can upload the project under test, run SuMo, and receive the results within the same VSCode workspace. Similar to tools like *coverage-gutters*[1], the extension displays mutation results directly on the Solidity code, making it easier for auditors to review mutations and understand their potential impact.

[1] vscode-coverage-gutters:`https://github.com/ryanluker/vscode-coverage-gutters`

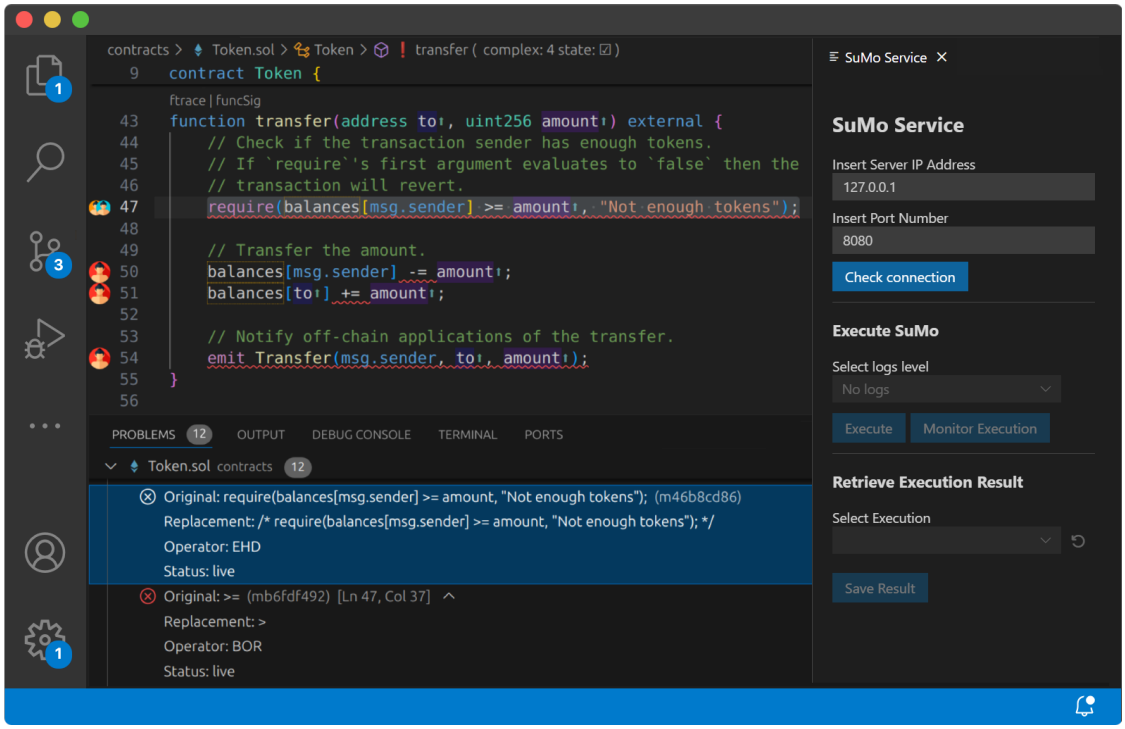


Figure 9.3: SuMo VSCode Extension with Service Configuration and Mutant Visualization Features

9.4 Validation

In the next, we discuss the experimental setup (Section 9.4.1), and the results of the simulations we performed to evaluate the proposed framework (Section 9.4.2). The validation centers around the diverse executors offered by the SuMo service and their impact on performance. The objective is to gain insights into how different parallelizatoin modes may help reduce the time needed for mutation testing within real-world software projects (Section 9.4.3).

9.4.1 Experiment Setup

To evaluate the SuMo service we ran simulations on three real-world Solidity projects with different scales and complexities. All the subjects, with the exception of the *MetaCoin* boilerprate, were selected from Code4Rena, as it provides realistic examples of contracts submitted by developers for a security audit:

- **MetaCoin**[2] is a compact template for token development;
- **ENS**[3] is a decentralized naming service to resolve a wide array of resources;
- **VTVL**[4] is a token management platform for founders of Web3-based projects.

[2]MetaCoin repository: `https://github.com/truffle-box/metacoin-box`
[3]ENS repository: `https://github.com/code-423n4/2023-10-ens`
[4]VTVL repository: `https://github.com/code-423n4/2022-09-vtvl`

In particular, for each project we performed the following steps:

1. **Local Execution**: First, we ran the stand-alone SuMo package on the project to establish a performance baseline;

2. **Service Execution**: Then, we executed five independent mutation testing runs on the project, one for each executor implemented in the service: OE, SCOE, SCME and DCOE and DCME. For this experiment, we fixed the maximum number of parallel containers spawned by each executor to 6. However, we ran additional simulations for the SCME and DCME executors, with variations in the number of parallel containers, to evaluate how the service performance responds to specific changes.

All simulations were performed on a Ubuntu Server 22 LTS machine with 8 cores, 16 threads and 16GB of RAM.

9.4.2 Discussion of Results

This Section compares the performance of the stand-alone SuMo tool against the SuMo service. Table 9.1 reports the total time required to complete a local mutation testing run on each project, and the average time needed to test a single mutant (Avg. TPM). It also shows the number of generated (#M), killed (#K) and live (#L) mutants, together with the achieved Mutation Score (MS). As can be seen, it took only 7 minutes to evaluate a simple boilerplate like MetaCoin. Instead, for a complex project with a large test suite like ENS, where SuMo produced over 3k mutants, the process required about 29 hours. Table 9.2 compares the performance of SuMo when executed locally and remotely, considering the different executors offered by the service: OE, SCOE, SCME, DCOE and DCME. For the latter, the Table details the time allocation for individual activities: 1) Extract Project, 2) Distribute Mutants, 3) Build Image and 4) Run SuMo. In the following we discuss the overhead introduced by each executor when performing preliminary operations, and their impact on the mutation testing time.

Table 9.1: Results for the Local SuMo Execution

Project	$\#M$	$\#L$	$\#K$	$MS(\%)$	$Time$	$Avg.TPM(s)$
MetaCoin	28	12	15	55.6	`00:07:07`	15.1
VTVL	336	131	191	59.1	`01:08:33`	12.2
ENS	3294	122	183	59.9	`29:42:22`	27.9

Overhead Time

While running SuMo is the most time-consuming activity of a mutation testing job, the service can introduce additional overhead while preparing the PUT for execution. Figure 9.4 compares the time required by each executor to perform these preliminary operations:

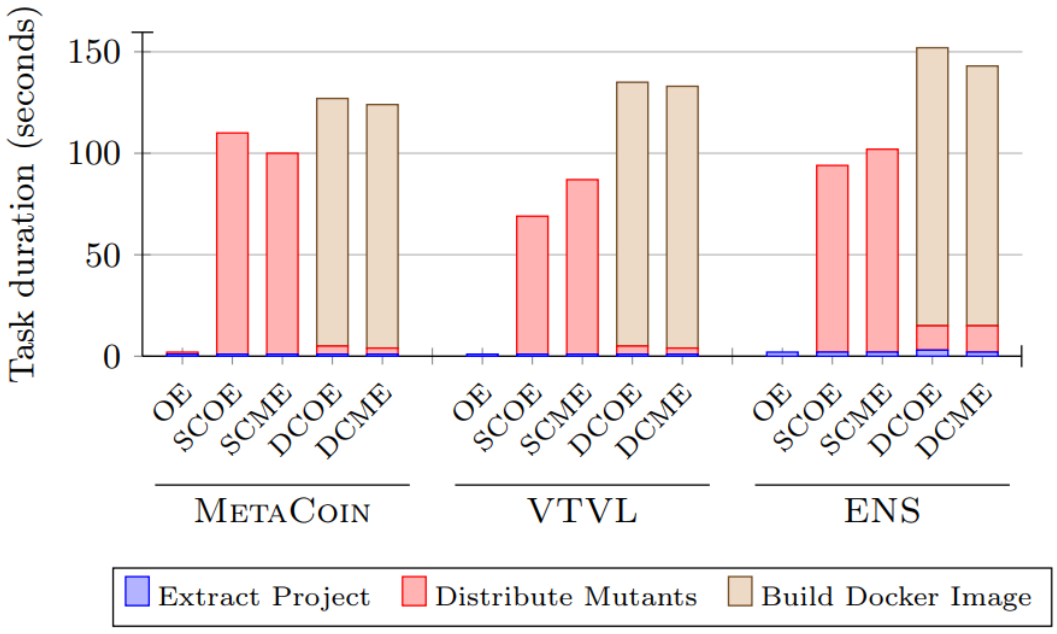
1. **Extract Project**: This task involves unpacking the PUT into the service's *workspace* directory. The time required for the project extraction phase is negligible across all projects and executors.

Table 9.2: Results for the SuMo Service Execution

Project	Task	Executor Type					
		Local	OE	SCOE	SCME	DCOE	DCME
MetaCoin	1) Extract Project	-	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
	2) Distribute Mut.	-	00:00:01	00:01:49	00:01:39	00:00:04	00:00:03
	3) Build Image	-	00:00:00	00:00:00	00:00:00	00:02:02	00:02:00
	4) Run SuMo	00:07:07	00:37:00	00:06:30	00:06:43	00:02:10	00:01:45
	TOT	00:07:07	00:37:02	00:08:20	00:08:23	00:04:17	00:03:49
VTVL	1) Extract Project	-	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
	2) Distribute Mut.	-	00:00:00	00:01:08	00:01:26	00:00:04	00:00:03
	3) Build Image	-	00:00:00	00:00:00	00:00:00	00:02:10	00:02:09
	4) Run SuMo	01:08:33	00:42:15	00:37:16	00:16:06	00:34:56	00:12:03
	TOT	01:08:33	00:42:16	00:38:25	00:17:33	00:37:11	00:14:16
ENS	1) Extract Project	-	00:00:02	00:00:02	00:00:02	00:00:03	00:00:02
	2) Distribute Mut.	-	00:00:00	00:01:32	00:01:40	00:00:12	00:00:13
	3) Build Image	-	00:00:00	00:00:00	00:00:00	00:02:17	00:02:08
	4) Run SuMo	29:42:22	11:06:42	11:31:41	07:34:26	11:49:27	07:58:53
	TOT	29:42:22	11:06:44	11:33:15	07:36:08	11:49:59	08:01:16

2. **Distribute Mutants**: This task involves the creation and execution of the *Configuration Container*. As explained previously, this component builds the list of mutants that will then be distributed to the parallel containers according to the specific strategy implemented by a given executor. Thus, this step is not applicable to OE, which simply enables a single operator per container. As shown in Figure 9.4, SCOE and SCME take considerably longer to complete this task than DCOE and DCME. This is because the former must install the project dependencies during this

Figure 9.4: Execution overhead of different Executors across projects



step, while the latter install dependencies while building the Docker image.

3. **Build Docker Image**: This task involves building a custom Docker image from a Dockerfile tailored to the PUT's requirements and installing its dependencies. Therefore, it is only applicable for the Dockerized executors, DCOE and DCME. It is important to note that the image must only be built once. Subsequent container creations do not necessitate image rebuilding, optimizing the execution workflow for DCME and DCOE.

SuMo Execution Time

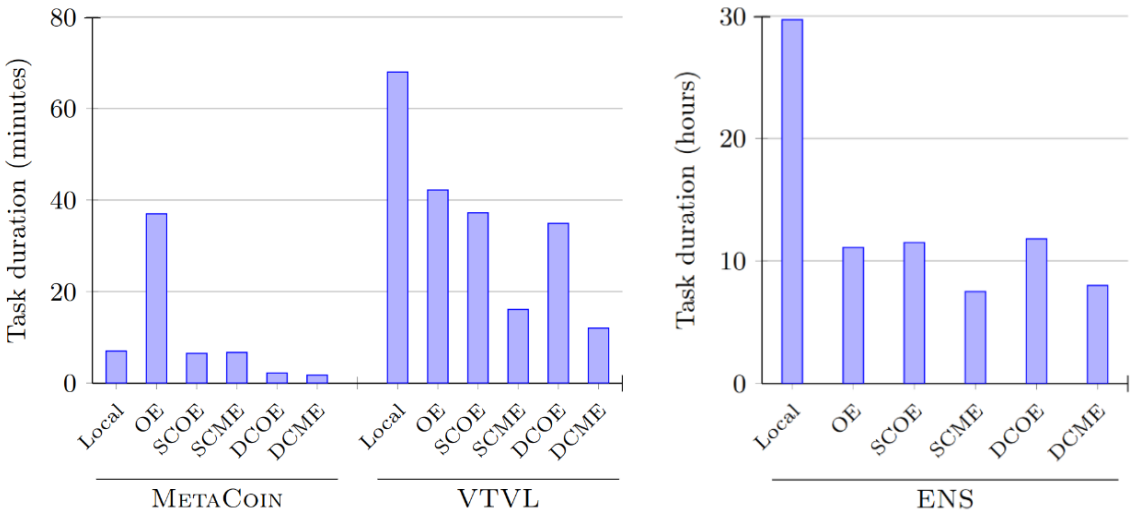
The time required to run mutation testing varies significantly across executors, with the Operators Executor (OE) taking the longest, and the Smart and Dockerized Combined Mutant Executor (SCME and DCME) showing significant efficiency (See Figure 9.5).

The **Operators Executor** (OE) proved to be rather inefficient due to its uneven distribution of mutants and repeated dependency installation process. This is evident in the MetaCoin project project, where the long time required for dependency installation negated the advantages of parallelism. In fact, the parallel execution took about five times longer than the local one.

The **Smart Combined Operators Executor** (SCOE) and **Dockerized Combined Operators Executor** (DCOE) increased efficiency by merging operators and distributing them across containers. However, they both showed limitations when handling larger projects. Indeed, it is not possible to optimally balance the distribution of mutants between containers using an operator-based grouping, making parallelization less efficient.

The effectiveness of the **Smart Combined Mutant Executor** (SCME) and **Dockerized Combined Mutant Executor** (DCME) lies in their capacity to evenly distribute mutants across Docker containers, contributing to a more efficient and balanced execution process. As Figure 9.5 shows, DCME outperformed SCME in most scenarios, **reducing the time** needed to run mutation testing **by 70%** with respect to the local SuMo execution

Figure 9.5: Time required to run SuMo on the PUT using Different Executors



(−57% for MetaCoin, −79.4% for VTVL and −73% for ENS). The SCME also achieved good time savings (−74.5% for VTVL and −74.5% for ENS), but it under-performed in smaller scale projects, to the point of increasing the execution time for MetaCoin (+17.1%) with respect to the local run. These two executors employ the same mutant distribution strategy, but they differ in their set-up processes and in the preparation of the containers for testing. DCME only needs to build the custom Docker image and install the project dependencies once, thus shortening the overhead time for subsequent mutation testing tasks. Instead, SCME uses a pre-built image, but is slowed down by the necessity to re-install all the dependencies every time a container is spawned to run SuMo on the PUT. This makes SCME less efficient in case of projects with a lengthier dependency installation such as MetaCoin and VTVL.

Table 9.3 compares the mutation testing duration of the two best executors, SCME and DCME, under various degrees of parallelism: 4, 6 and 8 Docker containers. While increasing the number of containers from 4 to 6 generally reduced testing time, further increasing the count to 8 did not yield proportional improvements, and even slightly extended the job duration for most projects. This plateau can be attributed to the limitations of the hardware used for running the simulation, but in any case a higher degree of parallelism may not always lead to performance improvements. To make the mutation testing process efficient and scalable, it is important to introduce a dynamic allocation strategy that adjusts the number of containers based on available resources and workload demands.

Table 9.3: Time required to complete a Mutation Testing Job using the SCME and DCME Executors with 4, 6, and 8 parallel containers

Project	**SCME**			**DCME**		
	4-Cont.	6-Cont.	8-Cont.	4-Cont.	6-Cont.	8-Cont.
MetaCoin	00:07:02	00:08:23	00:10:05	00:04:14	00:03:49	00:04:13
VTVL	00:20:41	00:17:33	00:17:44	00:18:06	00:14:16	00:12:40
ENS	08:58:20	07:36:08	07:57:34	09:21:21	08:01:16	08:14:36

9.4.3 Optimal Executors for Mutation Testing

The choice of the most suitable executor for mutation testing must be guided by project-specific considerations. These include the size and complexity of the project, the time required for dependency installation, and the availability of resources. The two top-performing executors in this context are Smart Combined Mutants Executor (SCME) and Dockerized Combined Mutants Executor (DCME). Both efficiently distribute mutants across Docker containers, making them a valid solution for large-scale projects that would normally result in long, expensive mutation testing runs. Notably, DCME has demonstrated a significant reduction in time consumption compared to local execution (-70%), closely followed by SCME. However, both executors can be particularly suitable in specific situations.

The **Smart Combined Mutant Executor** (SCME) utilizes a pre-built, optimized Docker image, which helps to conserve disk space, reduce build time and minimize re-

source usage. Its lightweight nature makes SCME especially suitable in scenarios with limited resources, such as constrained CPU or memory. However, in the MetaCoin project, lengthy dependency installations canceled out the advantages of parallelism, resulting in longer testing times compared to local, sequential execution. On the other hand, the **Dockerized Combined Mutant Executor** (DCME) leverages Docker's image management and dependency caching, significantly reducing the overhead of repeated dependency installations. This makes DCME particularly effective for projects with extensive dependency requirements. At the same time, building a custom Docker image may require significant resources, such as disk space and CPU, especially for large or complex applications.

Ultimately, the time savings achieved by SCME and DCME are very similar for projects that generate large amounts of mutants. Understanding the strengths of each executor can guide developers to make informed decisions that best align with the project's characteristics and available resources.

9.5 Related Work

Cost-reduction has long been a critical focus in mutation testing. Our work proposes a framework that speeds up the process through the parallel execution of mutation testing tasks and jobs within Docker containers. This Section reports related work close to our proposal.

Mutation Testing for Smart Contracts While there is a substantial body of research in mutation analysis for smart contracts [22, 3, 76, 40, 75, 100], no previous studies have proposed a comprehensive framework to streamline mutation testing for auditing activities. Over the past decade, various studies have focused on speeding up sequential mutation analysis, exploring techniques such as selective mutation, evolutionary algorithms, and higher-order mutation [141]. In the domain of smart contracts, *ContractMut* [75] implemented a manageable set of mutation operators, using selective mutation techniques to limit the number of generated mutants. The *Vertigo* tool [76], used mutant sampling technique to reduce the number of mutants to be run against the test suite. Instead, the *ReSuMo* [24] framework leveraged the outcomes from previous program revisions to incrementally update mutation testing results, thereby accelerating the mutation testing process for evolving projects.

Parallel Processing in Mutation Testing Parallel processing has also played a significant role in advancing mutation analysis. Historically, the concept of parallelizing mutation analysis locally has been recognized since the early 1990s [125]. Reales and Polo [110] later demonstrated that executing mutants in parallel can significantly reduce testing time, with the reduction scaling with the number of nodes employed. Li et al.'s work [98] extended this approach to Ruby programs utilizing cloud infrastructure for parallel execution. Closer to our proposal, Merkel and Georgeson [114] proposed a cloud-based distributed mutation testing system for Java programs, which addresses performance limitations by distributing tasks across Amazon EC2 computing nodes. Their system optimizes the distribution of mutation tasks by allocating equal-sized batches of

mutants to each worker node, proving this approach to be most effective for workload distribution. More recent work by Canizares et al. [37] improved the performance of mutation testing for large-scale systems by parallelizing mutant compilation and equivalent mutants detection, and proposing an advanced load distribution algorithm.

9.6 Directions for Future Research

Future work should focus on several key areas. Firstly, the framework should be evaluated on a broader spectrum of real-world projects to better asses its efficacy under different conditions. To ensure optimal resource usage, future enhancements should introduce the orchestration of mutation testing jobs across multiple nodes and dynamically adapt the size of the container pool to meet project-specific requirements and available infrastructure.

Inspired by practices in industrial settings, such as Google's use of developer feedback [137], we also plan to expand the implemented service to track the auditor's feedback on the productivity of live mutants. This information will help to develop heuristics for prioritizing interesting mutants for analysis, reducing the cognitive burden placed on auditors.

Finally, we plan to incorporate the use of large language models (LLMs) to simplify mutant analysis and the automated derivation of test cases.

Acknowledgements

This work has been supported by Quantstamp.

CHAPTER 10

ALCHEMIST: MUTANT-DRIVEN TEST GENERATIONS VIA LLMS

Bugs in Solidity smart contracts caused millions in financial losses only in the past year. Mutation testing can reveal deficiencies in test suites beyond those detected by simpler coverage techniques, but generating test cases to address surviving mutants is labor-intensive, limiting its adoption in high-stakes domains. To address this challenge, we introduce Alchemist, a novel framework for generating test cases from Solidity mutants using Large Language Models (LLMs). Alchemist enabling systematic refinement of test cases through iterative hypothesis generation and experimentation. Evaluation on Solidity projects demonstrates that this method can improves test quality over direct-prompting while reducing developer effort.

This chapter is based on material from the following publications: [17, 18]

* * *

10.1 Introduction

In blockchain ecosystems such as Ethereum, smart contracts are fundamental to the execution of decentralized applications and financial transactions. Failures in smart contract execution can have disastrous consequences. According to industry security reports [145], smart contract hacks in 2023 alone resulted in an estimated loss of $437.5 million. Amongst the most common causes of these breaches are bugs within smart contracts, such as rounding errors and insufficient validation. To mitigate these risks, thorough testing [20] and auditing are critical. Recognized as one of the most powerful methods for assessing test adequacy [4], *mutation testing* identifies gaps in test suites that may be overlooked by simpler coverage-based techniques [81], offering an additional layer of confidence before smart contract deployment. Even so, mutation testing remains underutilized in the domain of smart contract auditing. This is surprising, given its potential to enhance traditional auditing workflows. Auditors typically analyze code coverage as

part of their assessment to provide clients with feedback on the quality of their test cases. Thus, mutation testing could complement this process by offering deeper insights into test quality and enabling auditors to deliver more actionable recommendations.

One barrier to the adoption of mutation testing lies in the practical challenges associated with *analyzing surviving mutants* and designing test cases to kill them. Being a manual process, mutant analysis is a time-consuming and cognitively demanding task. For auditors, who are often required to complete smart contract reviews within constrained timelines, the responsibility of improving test suites generally falls on the client. Yet, auditors could enhance the value of their services by offering mutant-driven suggestions for new or improved test cases. To this end, it is essential to permit the automatic derivation of test cases from surviving mutants.

In smart contract development, automated test generation has received considerable attention, with several techniques already exploring the production of Solidity test cases (e.g., SynTest-Solidity [126], SolTG [32], and SolAR [55]). Nevertheless, no existing solution attempts to *improve the fault-detection capabilities* of an existing test suite by targeting live mutants, nor does so with the assistance of Large Language Models (LLMs).

Mutant-driven test generation via LLMs has shown promising results in other domains (e.g., MuTAP in Python [48]), but existing methodologies remain largely static. Prompts are issued once per mutant, with little to no reasoning or adaptation across iterations. Transferring the success of these techniques to Solidity smart contracts is therefore challenging. Solidity test suites are heterogeneous and structurally intricate, as they interact with complex contract states and permission structures. As a result, generating syntactically correct and semantically meaningful test becomes considerably difficult, making static, single-pass prompting approaches insufficient.

Recent studies suggest that LLMs perform better on complex tasks when guided by structured reasoning techniques, such as Chain-of-Thought [189] and iterative refinement [148]. In standard test generation, *ChatTester* [196] leverages ChatGPT to iteratively improve the correctness and quality of Java unit tests. Unlike earlier methods that rely on single-turn prompts, it decomposes the task into inferring the intention of the focal method, and generating the test using this natural language intention. This improves semantic alignment and reduces assertion errors. In automated program repair, AutoSD [89] shows that *hypothesis-driven reasoning* enables LLMs to produce more effective patches for buggy Python code. Drawing on the principles of *scientific debugging* [197], AutoSD mirrors the way human developers hypothesize, experiment, observe, and draw conclusions about the root causes of faults.

Building on these insights, we propose Alchemist (Automated LLM-based Creation of Hypotheses and Experiments for Mutant-driven Iterative Scientific Testing), the first framework for generating test cases from Solidity mutants via LLMs. Unlike traditional approaches, Alchemist embeds the scientific method into the code generation process, systematically refining test cases through iterative hypothesis generation, experimentation, and observation. This structured approach can target live mutants more effectively but also yield more transparent and explainable test generation steps.

In this study, we evaluate the effectiveness of Alchemist in enhancing test quality on projects submitted to *Quantstamp*[1] for a security audit. The results show that using a

[1]Quantstamp: `https://quantstamp.com`

hypothesis-driven approach can outperform direct prompting strategies, both in the generation of syntactically correct and mutant-killing test cases. Moreover, while Alchemist focuses on test generation for Solidity smart contracts, its methodology is general and can be extended to other software domains.

The rest of this chapter is structured as follows. Section 10.2 presents the research objectives. Section 10.3 introduces the proposed approach and framework, describing each phase of the mutant-driven test generation workflow. Section 10.4 presents the experiment methodology and setup, including the configuration of the LLM model, and the projects under test. Section 10.5 discusses the results of the experiments, presenting the improvements brought by the proposed approach. Section 10.6 outlines the threats to the validity, while Section 10.7 reports related work. Lastly, Section 10.8 draws directions for future research.

10.2 Research Objectives

The work presented in this Chapter aims to explore and validate the potential of combining Large Language Model (LLM)–based code generation with a Scientific Method workflow in the specific domain of smart contract testing. Our high-level objectives are:

1) Assess the impact of embedding the Scientific Method in test generation: The proposed approach embeds iterative *hypothesis generation*, *experimentation*, and *observation* in LLM prompts to guide the generation of test cases. We plan to explore whether this workflow does outperform direct prompting approaches. Key performance indicators include 1) the proportion of **correct test cases**, defined as tests that pass on the original, unmutated code, and 2) the proportion of **killing test cases**, defined as tests that successfully identify and expose relative mutants. This assessment will provide insights into whether applying structured scientific reasoning in LLM-driven workflows enhances test generation outcomes.

2) Evaluate the effectiveness of generated test cases against different types of mutants: This objective focuses on identifying which types of code mutations are most and least susceptible to automated test generation. To achieve this, we will analyze the performance of the generated test cases against mutation operators implemented in SuMo. In this way, we aim to pinpoint fault types that Alchemist is more apt at detecting, guiding the application of the technique in a practical context. At the same time, understanding characteristics of faults that are more difficult to generate test cases for can inform future improvements in LLM prompting approaches.

3) Investigate the benefits of Alchemist for smart contract auditors: This objective examines how leveraging LLMs to generate and refine Solidity test cases from surviving mutants can aid smart contract auditors. Given that manual mutant analysis for deriving test improvements is often time-intensive and cognitively demanding, this study evaluates the practical utility of automating such processes. Specifically, we aim to assess the extent to which the generated test cases improve the quality of smart contract test suites, as measured by the Mutation Score. Furthermore, we measure the associated

costs of using LLM-driven workflows, measured in terms of time and API usage.

10.3 The Alchemist Framework

Mutation testing activities focus on improving a test suite based on insights provided by surviving mutants. However, writing test cases to deal with live mutants is a complex task that requires substantial effort and time. Thus, we frame mutant-driven test generation as an iterative, hypothesis-driven process inspired by scientific debugging [197], and we introduce Alchemist, a framework that guides LLMs in systematically eliminating Solidity mutants. In Section 10.3.1 we present an example to clarify why writing test cases for Solidity smart contracts, both manually and via LLMs, can particularly challenging. Next, Sections 10.3.2 and 10.3.3 outline the overall methodology and architecture of Alchemist. Section 10.3.4 describes the complete test generation workflow in detail, and Section 10.3.5 concludes with an illustrative example of a conversation where Alchemist attempts to generate a mutant-killing test case.

10.3.1 Motivating Example

Live Solidity mutants can be complex to kill without deliberate reasoning. Consider the example in Listing 10.1. This function allows a user to withdraw funds from the smart contract, but only if they have enough balance. The `onlyHolder` modifier further restricts this operation to users with a custom *holder* role in the contract. A common mutation changes the comparison operator from ≥ (on line 3), to > (on line 4). At a glance, the mutation seems minor, but it introduces a significant behavioral difference: users can no longer withdraw their entire balance. To kill the mutant, a test should simulate the scenario where a user tries to withdraw all their balance (i.e., `amount` is equal to `balances[msg.sender]`), and expect the transaction to succeed. However, writing a test to detect this simple fault requires more than calling the function:

1. Before calling `withdraw`, the test suite must initialize the `balances` mapping to ensure that at least one user with *holder* role has some deposited funds.

2. In Ethereum, each transaction is executed in the context of a sender address. The test must invoke `withdraw` from an address with *holder* role.

3. The test must call `withdraw` with an amount equal to the *holder*'s current balance and check that the transaction succeeds. This may involve checking that no transaction revert occurred, and confirming that the *holder*'s balance in the contract is correctly decreased.

While the example above is relatively simple, real-world contracts often involve more complex logic. They typically implement access control mechanisms with multiple user roles, and interact with external contracts that must be correctly deployed and configured during testing. Moreover, a contract's observable behavior depends on various factors, such as the specific error-handling mechanisms used, the emission of events, and the presence of view functions (i.e., getters for public state). Each of these elements has different implications for how tests are written and how failures are detected. Mutations

```
1 function withdraw(uint amount) public onlyHolder{
2
3  ⊙ require(balances[msg.sender] >= amount);
4  ⋆ require(balances[msg.sender] > amount);
5
6   balances[msg.sender] -= amount;
7   payable(msg.sender).transfer(amount);
8 }
```

Listing 10.1: Simple Solidity Function to withdraw Funds from a Smart Contract

can impact subtle aspects of control flow or internal state that only manifest under specific transaction sequences or edge-case inputs. On top of this, testing frameworks like Hardhat [124] and Foundry [63] impose their own languages, conventions and libraries, all of which must be respected to produce valid and meaningful test cases. Thus, effective test generation cannot rely on surface-level code patterns, but requires contextual awareness and deep reasoning about contract behavior and test semantics.

10.3.2 Test Generation via LLMs and the Scientific Method

Recent work in LLM-based test generation, such as ChatTester [196], has shown the effectiveness of iteratively prompting LLMs based on feedback from test execution. Building on this foundation, our approach formalizes the process of mutant-driven test generation through the lens of the scientific method. This structured formulation introduces a principled framework for generating mutation-aware tests, where the reasoning behind each generated test is made explicit and traceable.

The process involves four key steps:

1. **Hypothesis**: Begin by formulating a *hypothesis* to explain why the existing test suite failed to detect the mutant. The hypothesis predicts the characteristics of a new test case that might successfully detect the mutant's fault.

2. **Experiment**: Based on the hypothesis, design and implement an experiment in the form of a new *test case*.

3. **Observation**: Execute the test case against the mutant and observe the results.

4. **Conclusion**: If the mutant is successfully killed by the new test case, the test is deemed effective and can be incorporated into the original test suite. If the mutant survives, the collected data is used to refine the initial hypothesis. This revised hypothesis informs subsequent iterations of the process.

This structure brings several advantages to mutant-driven test generation. First, it *sharpens the LLM's objective* by grounding each test in a specific hypothesis, resulting in more targeted and mutation-aware test code. Second, it introduces *adaptivity* through concrete observations: when a test fails to kill a mutant, the LLM is guided by execution feedback to revise its assumptions and generate more precise follow-ups. Finally, each

generated test is backed by a *reasoning chain* that can help developers and auditors alike understand why the test was written and how it contributes to the initial test suite.

10.3.3 Implementation of Alchemist

We have implemented Alchemist [117] as a Python-based toolchain that automates the test generation process using LLMs. The framework includes the following three key components, which we illustrate in Figure 10.1.

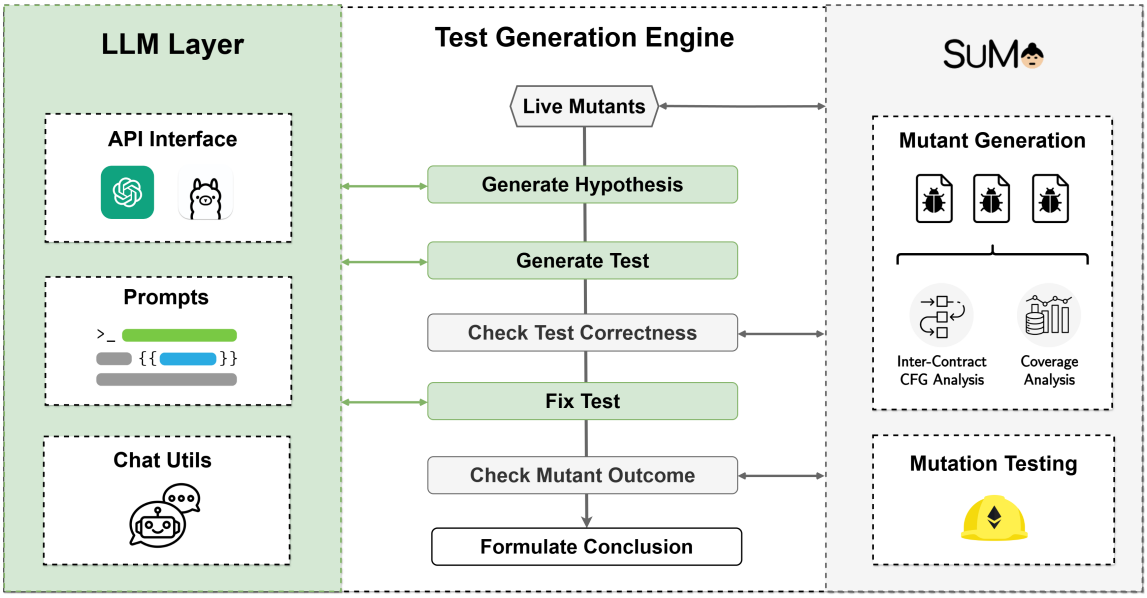


Figure 10.1: High-Level Architecture of Alchemist

Test Generation Engine The test generation engine orchestrates the entire hypothesis–experiment–observation–conclusion cycle by managing interactions between the LLM and the mutation testing tool. Specifically, it relies on the LLM for tasks related to hypothesis and code generation. Instead, it interacts with the mutation testing tool to handle tasks related to mutation testing and observing test outcomes.

LLM Layer This layer hosts all prompt templates and logic for interacting with LLM-based services, together with utilities for managing the conversation and its history. It serves as an abstraction over the underlying LLM provider, making it possible to switch between different models.

Mutation Testing Layer This layer includes logic for running mutation testing related tasks. To this end, Alchemist relies on SuMo. Within the proposed approach, SuMo plays two key roles. First, SuMo generates mutants of the target codebase. As we better explain in Section 10.3.4, we have extended SuMo with features for AST (Abstract Syntax Tree), CFG (Control Flow Graph), and test coverage analysis so that each mutant is enriched with information to build effective prompts. Second, SuMo provides the means to run experiments and observe their results as part of the test generation pipeline. Specifically,

its `pretest` functionality allows the engine to verify whether a newly generated test case (hypothesized by the LLM) is syntactically valid and passes against the original smart contracts. Instead, the `test` command runs the generated test file against a mutant. By observing the outcome of this process, Alchemist can determine if the mutant was killed (i.e., the hypothesis was correct) or survived testing (i.e., the hypothesis must be refined).

10.3.4 Test Generation Process

This section describes the test generation workflow implemented by Alchemist, following the scientific framework from Section 10.3.2. As shown in Figure 10.2, the process begins by iterating over the *live mutants* identified in the System Under Test (SUT). To ensure each mutant is analyzed independently, any conversation history with the Large Language Model (LLM) is cleared after each mutant is processed. In the following, we describe each step in the test generation process:

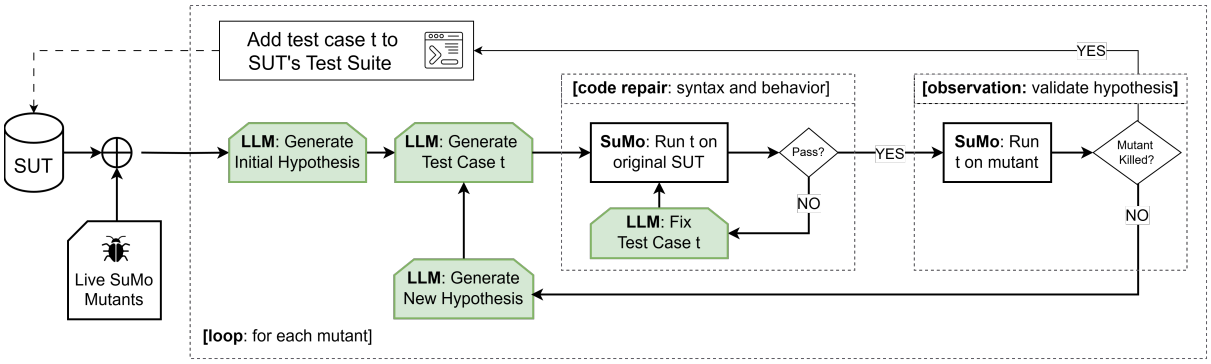


Figure 10.2: The Scientific Test Generation Process using Live Mutants

1) LLM: Generate Initial Hypothesis

In the first step, Alchemist prompts the LLM to explain why a test suite might have missed a live mutant, and hypothesize a test case that could detect it. The prompt is built using Chain-of-Thought (CoT) to increase the likelihood of generating a logically sound hypothesis [189]. CoT encourages the model to consider all aspects of the problem at hand [59], reducing the chance of logical leaps in the final answer. The prompt includes two key inputs:

- **Mutant**: A diff-based description of the live mutant;
- **Smart Contract**: A filtered view of the smart contracts in the SUT. This view isolates only the parts of the code that are likely relevant for understanding the mutation's impact and for generating a meaningful test.

The relevant contract code for a mutant is extracted as follows:

1. *Inter-Contract Control Flow Analysis*: We build the CFG of all smart contracts in the SUT using *Surya* [44]. Then, we identify all the functions and modifiers that can reach, or are applied to, the mutated function.

2. *Relevant Code Extraction:* We analyze the AST of each contract to extract these relevant functions and modifiers.

This approach ensures that the LLM is guided by only contextually relevant information, avoiding unnecessary distraction and optimizing token usage on complex projects. Additionally, cross-contract analysis ensures that the LLM has access to all relevant code, such as modifiers that may apply to the mutated function but are defined in other contracts.

2) LLM: Generate Test Case

In the second step, Alchemist prompts the LLM to generate a test case based on the hypothesis. Once again, it uses CoT prompting to guide the LLM's reasoning, providing it with prior chat messages for context, and:

- **Test File**: A filtered view of a test file from the SUT, containing test setup code and representative test methods (if any).

Even in this case, we do not provide the full test file to the LLM. Instead, we only extract relevant code from the file to avoid distractions and reduce token usage. The test file selection and extraction process works as follows:

1. *Test Coverage Collection*: We obtain the test coverage matrix for the SUT using *solidity-coverage* [150]. For every statement in a smart contract, this matrix lists all the test methods that exercise them.

2. *Selection of the Most Relevant Test File*: We identify the test file that covers the mutated statement the most times. If the mutated statement is uncovered, we consider the mutated function, and eventually we fall back to file coverage.

3. *Extraction of Test Setup and Relevant Test Methods:* From the selected test file, we extract only the test methods that exercise the mutated function, along with any global test variables, hooks, and setup code.

Generating Solidity test cases from scratch can be complex due to Ethereum's requirements for interacting with smart contracts, including deployment and account configurations. This process provides the LLM a reference for how existing test methods interact with the mutated function. Additionally, it grounds the LLM in the structure and setup of an existing test file, improving the likelihood of generating a syntactically valid and semantically meaningful test case.

3) SuMo: Run Test Case on original SUT

Sometimes, the test code generated by the LLM may contain syntax errors or incorrect assertions. To recognize this scenario, Alchemist runs the test on the original version of the SUT using SuMo.

4) LLM: Fix Test Case

If the generated test fails on the original version of the SUT, Alchemist prompts the LLM to attempt a fix. The prompt includes the following inputs:

- **Test Code:** The failing test code;
- **Contract Code:** The relevant sections of the contract, as defined in the Hypothesis Generation step;
- **Error Message:** The output from HardHat describing why the test failed.

Because this fix task is independent of hypothesis and test generation, the LLM is prompted starting with a clean history. This fix cycle continues until the code runs successfully or a maximum number of attempts is reached.

5) SuMo: Run Test Case on Mutant

If the generated test passes on the original SUT, Alchemist runs it on the mutant to validate the hypothesis. If the mutant is killed, the test case is added to the original test suite, and the process concludes.

6) LLM: Generate New Hypothesis

If the mutant survives, Alchemist rejects the hypothesis and proceeds to generate a new one. During this step, it re-prompts the LLM with an explanation that previous hypotheses failed to generate a mutant-killing test case. The conversation includes a trimmed chat history containing only the messages directly related to the earlier hypotheses generation. Access to previous messages allows the model to consider past attempts and avoid repeating the same mistakes. The process then cycles back to *Generate Test Case*, repeating until the mutant is killed or a maximum number of attempts is reached.

10.3.5 Illustrative Example of a Conversation

To illustrate a typical conversation in the test generation workflow, consider the live mutant in Listing 10.2. This mutant modifies the operator in the `require` statement from == to >=. This fault allows the sum in the left hand of the expression (`aSplit + fSplit + daoSplit`) to be greater than the `maxSplit` value of 100. Since the original test suite failed to detect this fault, it must be enhanced to ensure more throrough testing of boundary values. Table 10.1 shows how Alchemist engages with the LLM to produce the mutant-killing test case. The conversation is divided into *steps*, each representing a message in the chat. In each step, the *role* identifies who is responsible for that action: Alchemist (ALC) sends prompts and the LLM responds to them. Instead, the *history* lists prior messages included in the chat context so far.

Initially, Alchemist asks the LLM to propose a hypothesis for why the mutant survived (Steps 1–3). The LLM offers:

> **Hypothesis-1**: "*A test case that provides splits summing to less than 100% should pass on the original contract (causing a revert), but should fail on the mutant (causing no revert), thus killing the mutant.*"

```
1  function updateSplit(uint256 Id, uint256 aSplit, uint256 fSplit,
       ↪ uint256 daoSplit){
2
3  ⊙   require(aSplit + fSplit + daoSplit == maxSplit, "Primary split must
       ↪ equal 100%");
4  ★   require(aSplit + fSplit + daoSplit >= maxSplit, "Primary split must
       ↪ equal 100%");
5  }
```

Listing 10.2: Example of Live Mutant

```
1  it("Should revert when primary splits are less than 100%", async
       ↪ function () {
2     await expect(core721.updateSplit(0, 0, 50,
          ↪ 40)).to.be.revertedWith("Primary split must equal 100%");
3   });
```

Listing 10.3: Test Case generated from Hypothesis-1

Listing 10.3 shows an extract of the test case generated from the hypothesis. This test successfully runs on the original SUT, but it does not kill the mutant. In fact, the assertion expects a transaction revert when the primary split (computed by summing $aSplit = 0, bSplit = 50, daoSplit = 40$) is less than 100. However, both the original and the mutated contract revert in this scenario.

Since this first hypothesis is flawed, Alchemist asks the LLM to refine it (Steps 6–7 in Table 10.1). This time, the LLM offers:

> **Hypothesis-2**: "*A test case that provides primary splits summing exactly to 100% should pass on the original contract (as it meets the requirement), but should fail on the mutant (since it would allow splits greater than 100%), thus killing the mutant*"

In this iteration, the LLM generates the test case shown in Listing 10.4. This test ensures that transactions succeed when the primary split values sum to 100% (line 2) but expects a revert if the values exceed that threshold (line 3). Thus, the test passes on the original contract but fails on the mutant, effectively killing it. Ultimately, this test is incorporated into the original test suite, strengthening it by validating the correct contract behavior in a boundary value scenario.

10.4 Experiment Methodology and Setup

In this Section, we present the experiment we set up considering the objectives introduced in Section 10.2. Our goal is to evaluate the added value of the scientific method when incorporated into LLM-based, mutant-driven test generation. We aim to investigate the generation of correct and killing test cases against a conventional prompting approach,

```
1 it("Should revert when primary splits are more than 100%", async
    ↪ function () {
2   await expect(core721.updateSplit(0, 50, 50, 0)).to.not.be.reverted;
3   await expect(core721.updateSplit(0, 60, 50,
      ↪ 0)).to.be.revertedWith("Primary split must equal 100%");
4  });
```

Listing 10.4: Test Case generated from Hypothesis-2

and for specific mutation operator types. Additionally, we aim to investigate the overall benefits of Alchemist in terms of test adequacy improvements and costs. In the following, we provide details about the experiment methodology (Section 10.4.1), the test subjects (Section 10.4.2) and the used Large Language Model (Section 10.4.3).

10.4.1 Methodology

To validate the proposed approach, we conducted a three-step experiment:

1. Initial Mutation Testing Run We began by running mutation testing on a set of Solidity projects to establish a baseline Mutation Score (MS), representing the proportion of mutants killed by the original test suite. The MS serves as the starting point for evaluating subsequent improvements in test adequacy. Mutants were generated using the complete set of SuMo operators, as described in our previous work [10].

2. Test Generation using the Scientific Method Next, we applied Alchemist to the remaining live mutants, following the scientific methodology illustrated in Figure 10.2. To balance token expenses and effectiveness, we limited each mutant to two hypothesis generation attempts and allowed one fix attempt per generated test case. Correct tests that successfully killed mutants were integrated into the original test suite. We then executed SuMo on the enhanced test suite to compute the new Mutation Score.

3. Test Generation using a Standard Approach Finally, we repeated the test generation process without employing the scientific method. In this alternative run, we provided Alchemist with identical inputs and hyperparameters. However, we omitted the structured *hypothesis-experiment* cycles, relying on direct prompting for test generation. For reference, the default approach follows precisely the workflow outlined in Table 10.1, while excluding hypothesis generation (Steps 2-3) and refinement (Steps 6-7). In cases where the generated test does not kill the mutant, the LLM can still attempt to generate a new one, but without reasoning on the natural language assumption.

10.4.2 Experiment Subjects

We conducted the experiment on three Solidity projects submitted to *Quantstamp* for security auditing, all of which included test suites developed with HardHat. This decision was motivated by the prevalence of HardHat in Solidity development. In Table 10.2, we

Table 10.1: Example of a Conversation Workflow Resulting in a Successful Mutant Kill

Step	Role	Description	History
①	[ALC]	Describe Test Engineer Role	-
②	[ALC]	Generate Hypothesis {Mutant, Contract}	①
③	[LLM]	Return Hypothesis	①, ②
④	[ALC]	Generate Test Case {Test Setup}	①, ②, ③
⑤	[LLM]	Return Test Code	①, ②, ③, ④
`Run SuMo pretest <Test Code>: Pass`			
`Run SuMo test <Test Code>: Mutant Live`			
⑥	[ALC]	Generate New Hypothesis {Previous Hypothesis}	①, ②, ③
⑦	[LLM]	Return New Hypothesis	①, ②, ③, ⑥
⑧	[ALC]	Generate Test Case {Test Setup}	①, ②, ③, ⑥, ⑦
⑨	[LLM]	Return Test Code	①, ②, ③, ⑥, ⑦, ⑧
`Run SuMo pretest <Test Code>: Pass`			
`Run SuMo test <Test Code>: Mutant Killed`			

provide details about the projects supplied by the company. Due to the sensitive nature of the data, we identify each subject with a unique ID. We also report the number of contract files and test methods in the codebase with their statement and branch coverage. All test suites are written in Typescript, which is the target language for code generation.

10.4.3 LLM Configuration

The test generation framework is designed to be model-agnostic, offering the flexibility to integrate various LLMs. For this study, we chose the *GPT-4o-mini* model due to its computational efficiency, cost-effectiveness, and ability to handle mixed prompts containing both natural language and code snippets. Compared to other models, such as *Llama 3.1 8b*, it also provides a significantly larger context window, enabling more coherent

Table 10.2: Experiment Subjects

ID	Smart Contracts	Test Methods	Stmt. Cov.	Branch. Cov.	Test Language
1	15	99	80	63,3	Typescript
2	6	131	95,4	71,5	Typescript
3	3	165	98,3	91,3	Typescript

Table 10.3: Mutation Testing results for the original Test Suite

ID	Generated Mutants	Valid Mutants	Killed Mutants	Live Mutants	Mutation Score
1	1293	1201	583	**618**	48,5
2	1540	1446	901	**545**	62,3
3	541	503	418	**85**	83,1

extended interactions. Since generating correct tests is crucial, we lowered the model's default temperature to 0.1. Meanwhile, to avoid repeating incorrect outputs and to allow the model to adjust its assumptions while exploring multiple plausible solutions, we set the top-p value to 0.9. We used *GPT-4o-mini* out of the box, without any additional fine-tuning specific to this task or adaptations for Hardhat-related projects.

10.5 Discussion of Results

In this section, we present and interpret the findings from our empirical study, structured according to the objectives introduced in Section 10.2. Specifically, we examine (1) the comparative performance of the scientific approach versus the standard prompting approach, (2) the practical benefits of Alchemist for auditors, and (3) the results of the test generation process with respect to specific mutation operators. For context, Table 10.3 shows the outcome of the initial mutation testing run on the selected projects. For each project ID, the Table reports the total number of live mutants that were fed to Alchemist for generating test cases, and the Mutation Score achieved by the evaluated test suites.

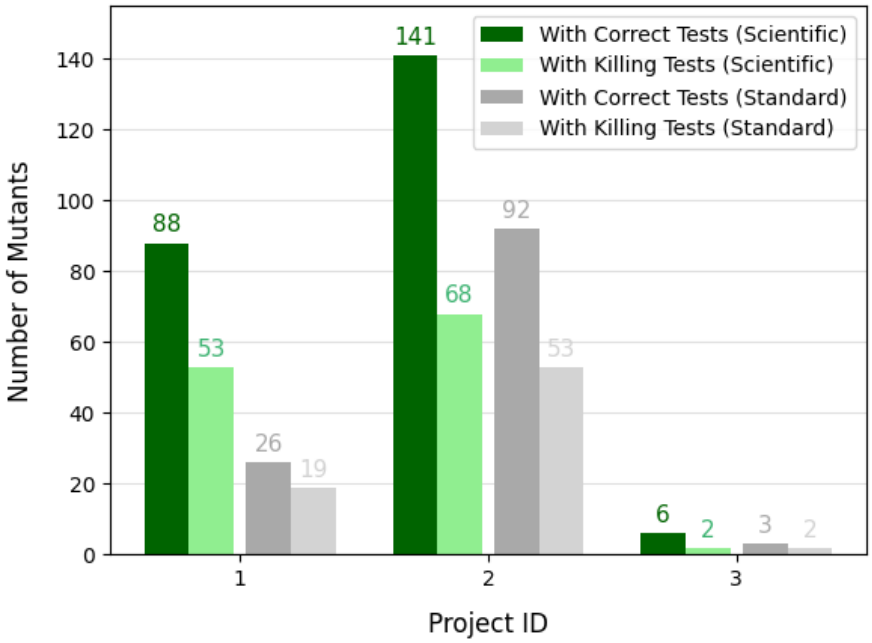
10.5.1 Impact of the Scientific Method on Test Generation

The experiment we conducted provides insights into the effectiveness of incorporating iterative hypothesis and code refinement into the test generation process. Figure 10.3 compares the scientific approach implemented by Alchemist with a standard prompting approach. In both cases, the Figure shows the total number of mutants for which the model generated **correct tests** (i.e., tests that run and pass on the original system under test), and the number of mutants for which the generated tests actually detect the fault (**killing tests**). Overall, the scientific approach outperforms the standard approach in all projects, both in terms of generating correct test cases and in detecting mutants.

Generation of Correct Test Cases The standard prompting approach generated 121 correct test cases from the live mutants of all projects. Embedding the scientific method in the test generation workflow produced almost twice as many, with a total of 235 correct test cases across projects. These results confirm that hypothesis-driven reasoning can reduce ambiguity in the test generation task and guide the LLM in producing code aligned with the contract's context.

While the scientific approach is indeed beneficial, it is important to note that generating working code remains a key bottleneck in test generation using LLMs. On average,

Figure 10.3: Results of Mutant-Driven Test Generation for the Scientific and the Standard Prompting Strategies



19% of live mutants were linked to a correct test case when using the scientific approach (with project 2 reaching 26%). Instead, the ratio of mutants with correct tests was consistently lower (below 10%) under the standard prompting strategy. The difficulty in generating working code is consistent with prior work on LLM-driven test generation [196, 129]. However, this challenge is amplified by Solidity's domain-specific nature and the intricacies of testing smart contracts.

Additionally, pre-trained LLMs tend to have less exposure to Solidity code compared to general-purpose programming languages [183], resulting in a narrower base of knowledge. This gap in expertise was observed in other studies that use pre-trained LLMs for generating smart contracts [41, 11], especially when the code exhibits more complex logic. Thus, although this initial result is promising, advanced techniques and fine-tuning might be required to boost the generation of correct test code.

Generation of Killing Test Cases Besides reporting the ratio of mutants with correct test cases, Figure 10.3 also shows the effectiveness of these tests in killing their target mutations. The fault-detection capabilities of the tests, once they are correctly generated, are encouraging. For the scientific method, 123 out of the 235 correct test ($\approx 52\%$) killed the relative live mutant. For the standard prompting approach, 74 out of the 121 correct tests resulted in a kill ($\approx 61\%$). In producing fewer valid tests, the standard approach may end up focusing on the more direct pathways that actually detect the fault, leading to a higher kill-to-correct ratio.

Still, the scientific method generates more killing test cases overall, expanding the original test suite in a more comprehensive manner. In both cases, among the mutants for which correct test cases were produced, more than half were killed. This shows that the

quality and fault-detection capabilities of the generated tests are strong when the model succeeds in producing valid outputs. Nevertheless, the success rate can vary considerably based on the complexity and visibility of the fault introduced by each mutation, a factor we further explore in Section 10.5.2.

Key Insights: *Impact of the Scientific Method on Test Generation*

Generating correct test cases for Solidity smart contracts is challenging, but the scientific method increases correctness rates by more than double. When the LLM generates correct tests, they are highly effective at detecting faults: over 50% of the correct tests kill their target mutants.

10.5.2 Analysis of Correct and Killing Tests by Operator

To understand which types of faults Alchemist can most easily detect, Table 10.4 reports the mutation operators for which at least one mutant was associated with a correct test.

Table 10.4: Generation of Correct and Killing Test Cases by Mutation Operator

ID	Description	Live Mutants	Correct Tests (%)	Killing Tests (%)	Killing / Correct Test Ratio
RRI	Force Role Revocation	11	63,6	0	0
ORFD	Remove Overridden Function	12	50,0	8,3	16,7
SVR	Replace String Literal	15	46,7	40,0	85,7
EED	Remove Event Emission	50	36,0	36,0	100
MOD	Remove Function Modifier	67	35,8	28,4	79,2
SKID	Insert or Remove Super Keyword	12	33,3	0	0
EHD	Remove Reverting Statement	94	33,0	30,9	93,5
CER	Replace Type Casting	31	25,8	0	0
PLR	Simulate Precision Loss	9	22,2	22,2	100
BOR	Replace Binary Operator	413	21,8	7,7	35,6
ISD	Remove Initialization Statement	5	20,0	0	0
NVR	Replace Number Value	221	9,5	3,6	38,1
RSD	Remove Return Statement	53	9,4	3,8	40
CSC	Modify Conditional Statement	69	7,2	2,9	40
AVR	Replace Address Value	56	7,1	5,4	75
GVR	Replace Global Blockchain Variable	18	5,6	0	0
MCR	Replace Global Math./Crypto Function	19	5,3	5,3	100

Generation of Correct Test Cases The data indicates that capacity to produce correct code is contingent on how transparent or complex the mutation is to the LLM. When a mutation operator modifies the contract in a way that the testing objective remains relatively straightforward, the LLM proposes test cases that are more likely to run and pass on the original (unmutated) contract.

For example, the ORFD (Overridden Function Deletion) operator and SVR (String Value Replacement) operators presumably retain a clear relationship to visible functionalities of the contract, such as overridden method logic or specific string parameters. EED (Event Emission Deletion), MOD (Modifier Deletion), and EHD (Exception Handling Deletion) mutations also yield correct tests at a moderate rate. This pattern may indicate that LLMs can sometimes generate the appropriate test scenario, particularly when the mutant's effect on the contract logic is conceptually straightforward (e.g., removing a `onlyOwner` modifier or a `require` statement will strip a function of some precondition checks). Still, the cascading implications for test code can be non-trivial, potentially requiring specific state and account settings to trigger the failing branch.

Instead, operators that introduce more complex modifications, such as altering global blockchain variables (e.g., the block `timestamp`) or contract addresses pose a greater challenge to the LLM's generative capabilities. These mutations often demand specific test steps such as interacting with external contracts or manipulating block properties. In such cases, even the inclusion and refinement of hypotheses might not be enough to infer necessary assertions or the correct environment setups, especially if these are not already present in the reference test code.

> **Key Insights:** *Analysis of Correct and Killing Tests by Mutation Operator*
>
> Mutation operators that introduce intuitive and clearly testable changes (e.g., event, exception handling, and modifier deletions) yield the highest mutant-killing rates. Mutations embedded in complex logic (e.g., global variable changes, role revocations) present a challenge, requiring more nuanced test designs.

Generation of Killing Test Cases A first clear observation is that operators with a high rate of correct test generation do not automatically translate to proportional success in mutant detection. When the LLM does generate correct test code, the likelihood of killing the mutant is promising, but this only seems true for specific types of mutations.

For instance, the RRI (Revoke Role Inseriton) operator forces a privileged account (e.g., an administrator) to give up its role, a fault that might seem conceptually straightforward to detect. Although live RRI mutants display a relatively high level of correct tests (63.6%), all of them fail to trigger the chain of events necessary to expose the underlying fault. In fact, killing a RRI mutant requires a test capable of both triggering the role revocation statement and subsequently invoking a privileged function from the same account, a scenario that might be complex for the LLM to infer.

On the other end of the spectrum are operators like EED, EHD, and MOD, which achieved high to perfect killing-to-correct test ratio. That is, whenever the generated test runs on the original contract without failure, it almost always detect the mutation for which it was generated. A plausible explanation is that these mutations alter externally observable behaviors that the LLM can more concretely detect and incorporate into the test logic. For example, omitting an event emission is straightforward to confirm by checking for missing logs, while disabling exception handling statements (which are also typically embedded in modifiers) can be detected by expecting a transaction revert. For the latter, the subtlety lies in generating tests that include boundary conditions or erroneous inputs, which is not trivial.

Table 10.5: Mutation testing results for the enhanced Test Suite

ID	Approach	Valid Mutants	Killed Mutants	Mutation Score
1	Scientific	1201	653 (+**70**)	54,4 (+**12%**)
	Standard		615 (+32)	51,2 (+5,5%)
2	Scientific	1448	994 (+**93**)	68,7 (+**10,2%**)
	Standard		987 (+86)	68,2 (+9,45%)
3	Scientific	503	418 (+2)	83,1 (+1,3%)
	Standard		418 (+2)	83,1 (+1,3%)

Mutations generated by PLR and MCR were also always detected when the LLM was able to generate correct test code. These mutations alter critical arithmetic pathways, such as simulating precision losses in calculations or introducing errors within hashing functions, leading to quantifiable discrepancies.

Overall, results show that when the LLM does generate correct test code, the likelihood of killing the mutant is promising. However, similar to test enhancement performed by human developers, LLMs more easily detect mutations whose effects are directly observable and require more intuitive test scenarios. Inferring necessary setups and test steps can be complex when the mutation is subtle or buried within complex logic. Improving kill rates may involve guiding the model at a more granular level, instructing it to further dissect the test code, transaction by transaction, and refine it through multi-turn reasoning.

10.5.3 Impact of Alchemist on Test Quality

One of the primary motivations of this work is to reduce the time and cognitive effort auditors spend when reviewing live mutants and provide test improvement tips to customers. Table 10.3 shows the outcome of the initial mutation testing run, including the Mutation Score achieved by the original test suites. Table 10.5 then reports the Mutation Score values once the killing tests generated by Alchemist are incorporated. In the case of the first two projects, the scientific approach leads to a sensitive increase in the Mutation Score (12% and 10.2%, respectively), confirming the tangible value of the new tests.

For the third project, which started with an already high Mutation Score (above 80%), the gains were predictably smaller (1.3%). Nonetheless, even minor improvements can be significant in high-stakes contexts. It is also worth noting that the time an monetary costs for deriving these test cases are relatively low monetary cost. Our experiments with GPT-4o-mini totaled about $2 USD across all projects when using the scientific prompting method. Execution time was also manageable: approximately 11 hours for the first project, 7 hours for the second, and 1 hour for the third. When considering the time and effort usually invested by developers in deriving tests manually, Alchemist can represent a valid aid for test suite enhancement. Future work might push these benefits further through domain-specific fine-tuning.

Key Insights: *Impact of Alchemist on Test Quality*

Applying the scientific method increases the Mutation Score by up to 12%, particularly in projects with moderate initial adequacy. The low cost of test generation makes this approach feasible for large-scale smart contract projects, reducing the burden on auditors while improving test quality.

10.6 Threats to Validity

In this Section, we discuss factors that may affect the validity of our results [147].

Threats To Internal Validity Internal validity refers to the extent to which our experimental setup accurately measures the impact of Alchemist on LLM-driven test generation. Several factors may influence the validity of our results. First, the accuracy of our results depends on the effectiveness of the mutation testing tool (SuMo) in generating meaningful mutants. If a large proportion of mutants are equivalent (i.e., they do not introduce functional changes), they could artificially lower the effectiveness of our approach. The hyperparameters selected for GPT-4o-mini, such as temperature (0.1) and top-p (0.9), may influence the quality of the generated test cases. Although these values were chosen based on prior work and empirical tuning, different settings could lead to variations in effectiveness. The results may be influenced by how effectively prompts are designed.

Threats To External Validity External validity concerns the generalizability of our findings beyond the tested Solidity projects. Our evaluation is based on three Solidity projects submitted for security auditing, all using the Hardhat framework. While these represent real-world projects, results may differ for contracts using different development environments and or testing languages (e.g., Python for Brownie). We conducted our experiments using GPT-4o-mini, a general-purpose language model not explicitly trained on Solidity. Thus, performance may vary with other models (e.g., Llama, Claude) or future iterations of GPT.

10.7 Related Work

Automated test generation has long been a critical focus in software testing. Our work investigates the intersection of LLM-driven test generation and mutation testing to automatically improve the quality of smart contract test suites. This Section reports related work close to our proposal.

Test Generation using LLMs Automated test generation with LLMs has attracted growing interest, particularly as models like ChatGPT have demonstrated promising abilities to synthesize unit tests directly from source code. Siddiq et al. [153] investigated the use of LLMs for generating unit tests for Java programs. While their study showed that models such as Codex could improve test coverage on certain benchmarks, they also

highlighted difficulties with complex codebases (e.g., redundant assertions and empty test cases). To address some of these shortcomings, Pan et al. [131] combined static analysis with LLMs to enhance unit test generation across multiple programming languages. Their approach employs targeted code analysis and transitions from one-shot prompting to iterative prompting, refining test cases over multiple interactions. Yuan et al. [196] introduced *ChatTester*, a framework that leverages ChatGPT to iteratively improve the correctness and quality of Java unit tests. ChatTester addresses common issues in LLM-generated tests, such as compilation failures, incorrect assertions, and poor usability. First, it prompts the LLM to infer the semantic intent of the method under test. Then, it enters an iterative refinement loop that compiles the test, captures error messages and execution feedback, and re-prompts the model with contextualized input. Building on the usefulness of iterative refinement, Alchemist extends this paradigm with a mutant-driven methodology tailored to smart contracts. Rather than relying solely on feedback from compilation or coverage metrics, it formulates a rationale for each surviving mutant and generates a test to kill it. This not only enhances fault-detection capabilities, but also produces more interpretable tests.

Test Generation for Smart Contracts Several tools target automatically generating test suites for Solidity smart contracts, primarily using coverage metrics as their driving criterion. *AGSolT* [54] combines random search and genetic algorithms to explore contract behavior, achieving high branch coverage across a range of real-world contracts. Similarly, *SolAR* [55] introduces a pipeline that combines genetic algorithms with fuzzing, optimizing for coverage while accounting for the unique operational patterns of blockchain transactions. Tools like *SynTest-Solidity* rely on metaheuristic strategies (random search, NSGA-II, MOSA) to maximize branch coverage. However, none of these tools incorporate mutation testing, which better approximates real-world fault detection. Alchemist addresses this gap by shifting the focus from maximizing coverage to eliminating surviving mutants.

Mutant-Driven Test Generation The concept of using mutants to drive test case generation has been explored for quite some time. Early work by Fraser and Zeller [65] introduced a mutation-driven test generation approach for object-oriented classes. This method could produce test suites that significantly outperform even high-quality, manually crafted ones in uncovering defects. Recently, Dakhel et al. introduced *MuTAP* [48], a framework that augments LLM prompts with information about surviving mutants to guide the generation of bug-revealing test cases. Results show that MuTAP significantly outperforms state-of-the-art tools like Pynguin, detecting up to 28% more faulty human-written code snippets. While MuTAP marks a significant step forward in combining LLMs with mutation testing, its methodology remains static. Prompts are constructed from structured metadata and issued once per mutant, with no reasoning or adaptation across iterations. Instead, *Alchemist* guides LLMs through an iterative process, allowing the model to explore why a mutant might survive, and revising its understanding based on runtime feedback.

10.8 Directions for Future Research

Ensuring the quality of test suites is a critical challenge, particularly in financial domains where smart contracts are widely adopted. This thesis introduced a novel LLM-based methodology that incorporates scientific method principles to systematically hypothesize and refine test cases from undetected Solidity mutants.

Despite the promising initial results, the study highlighted some challenges in LLM-driven test generation for smart contracts. Generating working tests remains the main bottleneck in automated test generation using LLMs. Future enhancements to Alchemist should include more advanced context modeling or fine-tuning to better capture domain-specific testing heuristics. Fine-tuning LLMs on a corpus of Solidity projects and associated test suites may enhance the model's familiarity with specific testing patterns and help generate valid tests.

Furthermore, as for human developers, LLMs are more likely to derive a killing test if the impact of the target mutation is intuitive and directly observable. Mutants demanding more intricate knowledge of Solidity's execution environment remain challenging to detect. Improving kill rates may require a more granular approach that guides the model to dissect and refine tests step by step.

Finally, we believe that the cross-language applicability of this approach should also be explored. While this research focused on Solidity, adapting the methodology to general-purpose language would provide insights into its effectiveness without considering domain-specific intricacies that may influence the results.

Acknowledgements

This research has been supported by Quantstamp.

PART IV

TRANSACTION REPLAY FOR SMART CONTRACT MAINTENANCE

CHAPTER 11

CATANA: REPLAY TESTING FOR UPGRADEABLE SMART CONTRACTS

Blockchain technology is increasingly adopted in scenarios requiring trust and data integrity. On the Ethereum blockchain, the proxy pattern has become increasingly popular because it allows smart contract code to evolve while preserving stored data. However, a key challenge remains ensuring that such upgrades do not introduce breaking changes or cause disruptions to other contracts and off-chain systems. In this chapter we introduce Catana, a framework that leverages historical transactions for Capture-Replay testing of proxy-based Upgradeable Smart Contracts (USCs). Catana assesses the potential impact of an upgrade by comparing the outcomes of replayed transactions with those from the previous version deployed on the main network. Additionally, it extracts and decodes contract state variables, providing deeper insights into how code changes affect the contract state, and helping developers mitigate issues before deployment. Experiments demonstrate that analyzing storage data accounts for the majority (about 86.5%) of detected disruptive upgrades. We also evaluate different policies for building replay test suites from historical transactions. Results identify a strategy that maximizes effectiveness while requiring a small number of replay test executions. Even a test suite containing just one transaction per each invoked method achieved good effectiveness (about 60%) in detecting disruptive upgrades.

This chapter is based on material from the following publications: [12, 16]

* * *

11.1 Introduction

Blockchain technology is widely known for its ability to store data immutably, ensuring that once data is recorded, it cannot be altered in any practical manner. Among

the existing blockchain platforms, Ethereum stands out due to its support for smart contracts, tamper-proof programs that can be deployed and executed on the blockchain [191]. This capability allows users to interact with decentralized applications (DApps) with confidence, knowing that the underlying logic cannot be arbitrarily modified.

Although the immutability of smart contracts fosters integrity and trust, it also presents a significant challenge: it prevents direct bug fixes and feature updates. To address this limitation, developers have introduced mechanisms for implementing Upgradeable Smart Contracts (USCs), with the *proxy pattern* emerging as a popular solution [34, 149]. This pattern decouples a contract's storage from its logic, allowing developers to release upgrades without redeploying or migrating stored data. Thanks to its convenience and simple design, the proxy pattern has since been formalized through community standards and is now supported by major frameworks such as OpenZeppelin [127].

While the proxy pattern is now considered technically mature, developers still approach upgrades with caution. The fact that only a small percentage of USCs have been updated more than once indicates concerns about the potential impact of the introduced changes [142]. One key contributing factor is the lack of proper support for regression testing. Developers primarily upgrade contracts to improve usability, typically by adding or modifying functionalities [101]. However, beyond the risk of introducing new bugs or security vulnerabilities [99], changes to the business logic of a smart contract can cause *compatibility issues*.

A large-scale study over 60M Ethereum smart contracts revealed frequent occurrences of ABI-breaking changes in proxy-based systems [99], where updates to the logic contract break compatibility with the interface. Another problem is given by the potential for storage collisions [146, 79], where new variables in the logic contract overlap with existing storage slots. Both issues are highly relevant, but more subtle changes that neither break the ABI nor corrupt storage can still affect smart contract behavior while being harder to detect. Such changes may introduce inconsistencies in contract outputs and states, possibly causing disruptions in dependent systems (e.g., in DApp front-ends, off-chain systems, or other smart contracts), and leading to unintended consequences for users.

Current tools offer only partial safeguards, as they mostly focus on preventing storage collisions. For example, OpenZeppelin's Upgrades plugin [128] includes useful predeployment checks to ensure storage integrity, which alone does not guarantee behavioral compatibility. To truly improve the reliability of USCs, more advanced testing approaches are needed. Fine-grained behavioral analysis, particularly one that accounts for real-world usage patterns, is essential to detect subtle but disruptive changes before upgrades reach production.

Capture-Replay testing is a testing technique that collects interactions between users and the system under test (SUT), leveraging them to create replayable test scripts [28, 73, 93, 80, 29]. While capturing execution traces in traditional systems can be intrusive and introduce performance overhead, public blockchains immutably and publicly log all user transactions by design. This offers a unique opportunity for non-intrusive, large-scale replay testing of blockchain applications. However, despite this potential, no existing framework is designed to detect behavioral incompatibilities introduced by smart contract upgrades. Prior work has explored historical transactions in other contexts, but with fundamentally different objectives. For example, SmartGift [205] demonstrates how real-world transactions can guide the generation of effective test cases, but it focuses on newly

created contracts similar to the ones deployed, not regression testing of upgraded ones. Other transaction-centric frameworks emphasize monitoring, attack detection, or runtime protection [193, 46, 202], without addressing upgrade compatibility.

This Chapter presents the first regression testing approach and framework for Ethereum Proxy-based USCs using historical transaction replay. The proposed approach consists of the following main steps:

1. Retrieving historical transactions to build a replay test suite;
2. Forking the Ethereum Mainnet and executing these transactions on the original USC to derive test oracles (i.e., the expected outcomes);
3. Simulating an upgrade and replaying the transactions on the new USC version;
4. Checking the consistency of the outcomes (i.e., the *output* of the transaction, and its impact on the Proxy *storage*).

We implemented this procedure in a reference framework called Catana (*Contract Assessment through TrANsaction replAy*). As part of its replay testing workflow, Catana analyzes not just the output of a replayed transaction but also its impact on the internal state of the proxy. This improves observability during testing and offers deeper insights into how contract upgrades affect both behavior and state. We made Catana open-source on GitHub to support developers in: *identifying disruptive changes* prior to deployment, *validating intentional changes*, and *debugging* eventual issues by analyzing pre- and post-upgrade logs. Catana is agnostic to the proxy pattern used, making it applicable to established upgradeability standards (e.g., Transparent and UUPS Proxies [127]).

With this study, we aim to answer the following Research Questions:

- **RQ1**: *How does storage data analysis impact the effectiveness of Capture-Replay testing for USCs?*
- **RQ2**: *Which transaction selection policy most effectively identifies disruptive behaviors in USCs?*

First, we investigate to what degree smart contract storage analysis improves Capture-Replay testing capabilities. The experimental results show that storage data analysis proved essential for flagging code changes that might affect the behavior of a USC, potentially leading to regression issues and compatibility concerns. Indeed, out of all the disruptive upgrades that Catana detected, about 86.5% were exclusively identified through changes in state variables. Furthermore, considering the high volume of transactions that might be associated with real-world USCs, we evaluate the effectiveness of four selection policies for building replay test suites. The results show that focusing on method-level diversity consistently outperforms other approaches in terms of fault-detection and the number of replayed transactions. Notably, even a test suite containing just one transaction per each invoked method achieved good effectiveness, detecting about 60% of disruptive upgrades.

Overall the main contributions of this thesis can be summarized as follows:

- A novel capture-replay testing approach for proxy-based USCs based on historical Ethereum transactions;

- Catana, an open-source and proxy-agnostic framework for capture-replay testing, which supports the detection of potentially disruptive changes in upgrades, validation of intentional changes, and debugging;
- An empirical assessment that estimates the impact of observability during replay testing through both output comparison and storage state analysis;
- The evaluation of four transaction selection policies, highlighting the trade-offs between the size of the generated replay test suites and their fault-detection capabilities;
- The formulation of a set of guidelines for practitioners summarizing the main insights we observed in the study.

The rest of this Chapter is organized as follows: Section 11.2 describes the design of Catana and its replay testing worklow in detail. Section 11.3 expands on the research questions, while Sections 11.4 and 11.5 describe the experiment setup and results. Section 11.6 discusses threats to validity of the study. Finally, Section 11.7 reviews related work and Section 11.8 identifies future research directions.

11.2 The Catana Framework

Upgrading smart contracts deployed on the Ethereum Mainnet is a delicate activity. Developers must be aware of how code changes impact contract behavior, whether the new code performs as expected and, most importantly, whether it might break existing functionalities. To illustrate how Catana assists in upgrade scenarios, let's consider a Proxy-based Upgradeable Smart Contract (USC) including a Proxy contract P and a Logic contract L_n. The Proxy features several transactions as it is being actively used by the community. Over time, the logic contract L_n must be upgraded to L_{n+1} to improve some functionalities or to fix bugs. Catana is designed to ensure the consistency of this upgrade process by verifying that selected transactions executed on USC(P, L_n) give the same output on USC(P, L_{n+1}). Moreover, Catana logs any changes in the smart contract storage that occur as a result of the upgrade. These logs provide developers with a detailed record of any storage modifications, allowing them to assess the impact of the code changes. Overall, Catana can support developers in the following ways:

1. **Identify Disruptive Changes**: Catana flags any discrepancy in transaction outputs or storage values. This makes developers aware of potentially disruptive changes and allows them to review code before deployment.
2. **Validate Intentional Changes**: When the upgrade is meant to alter certain behaviors (e.g., after introducing a bug fix), Catana 's logs confirm whether the modifications align with the developer's expectations.
3. **Support Contract Analysis and Debugging**: The pre- and post-upgrade state logs allow developers to more easily understand the impact of the modifications and determine the cause of eventual issues.

The tool is open source and can be found on GitHub [118]. In the following Section, we describe the replay testing process of Catana, while Section 11.2.2 provides details its storage analysis workflow.

11.2.1 Capture-Replay Testing Approach

Catana provides a fully automated test harness for replay testing, enabling developers to capture transactions from an original USC(P, L_n) and replay them on an upgraded USC(P, L_{n+1}) without having to manually create or configure test scripts. To this end, Catana requires access to:

- *Transactions to replay*: a set of transactions T that were executed on USC(P, L_n);
- *Deployed USC addresses*: The addresses of USC(P, L_n), as deployed on the Mainnet;
- *Upgraded USC sources*: The source code and ABI of the USC(P, L_{n+1}) under test.

Although the upgraded sources are assumed to be available locally, Catana automatically retrieves the addresses and historical transactions of USC(P, L_n) by interacting with external services such as Etherscan. Once these inputs are gathered, Catana sets up the test environment and begins the replay process. Under the hood, Catana leverages the Hardhat [124] framework and encapsulates its replay testing logic in a single custom test file. This file defines three parametric test methods, each targeting one transaction $t \in T$. The first method, T_{oracle}, runs the transaction t on USC(P, L_n) to derive the test oracles and stores them for future comparison. T_{replay} simulates the logic upgrade to USC(P, L_{n+1}) and replays t under the upgraded contract's logic, capturing the new test outcomes. Lastly, T_{outcome} checks the new outcomes against the oracles, reporting any discrepancies that might indicate regression bugs or alterations in behavior. The capture-replay workflow is illustrated in Figure 11.1, and in the following we describe each step in detail.

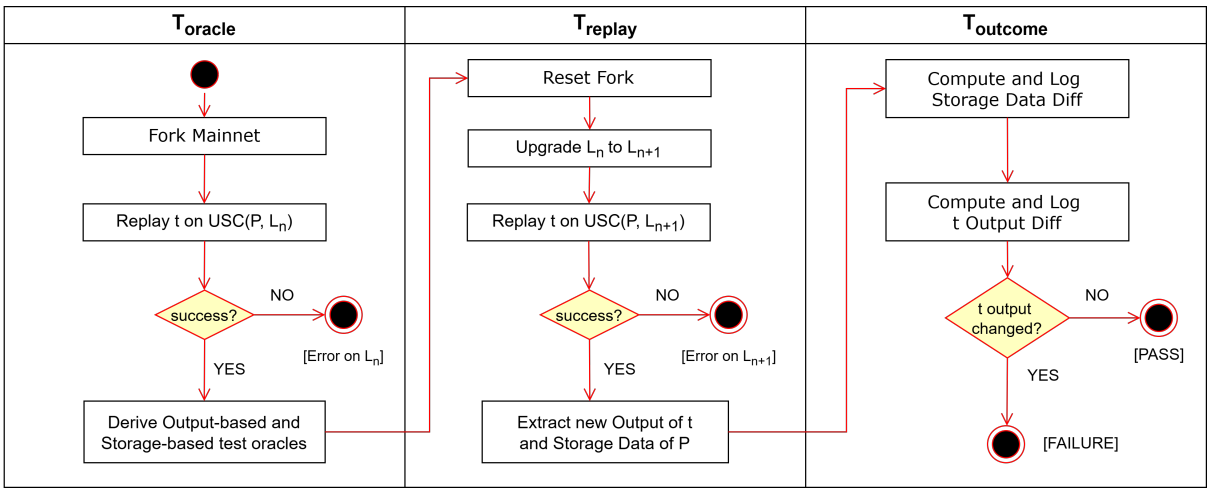


Figure 11.1: Capture-Replay Testing Workflow as implemented in CATANA

Step 1) $\mathbf{T_{oracle}}$ This test method automatically derives oracles for regression testing based on the behavior of the original USC(P, L_n). In particular, it captures:

- **Output-based test oracle**: the return value produced during a transaction t (if any);
- **Storage-based test oracle**: a snapshot of the contract's state variables after the transaction t completes.

To achieve this, the test begins by forking the Ethereum Mainnet at the precise block right before the transaction of interest t occurred. This ensures that the contract's state matches the one at the time of the original execution. By forking Mainnet, Hardhat retrieves the historical on-chain data and exposes it as though it were available locally, eliminating the need to mock external dependencies that might be invoked by t. Once the environment is forked, the test obtains the Proxy contract (P) and logic contract (L_n) from the forked network using their respective ABI definitions and deployed addresses. It then impersonates t's original sender address and replays the transaction on USC(P, L_n), applying the historical inputs and value (i.e., the amount of transferred funds). This replay involves two types of calls: (i) a *static call*, which simulates execution without modifying the blockchain state and allows Catana to capture the *output-based* oracle (including any return values or error messages), and (ii) a standard *transaction call*, which actually modifies the state. As shown in Figure 11.1 error scenarios are handled to ensure meaningful replays. If a call fails on the original contract (e.g., due to reverts, network time-outs, or unexpected errors) it is excluded from further replay testing. If the transaction call completes, Catana records the *storage-based oracle* by extracting state variables from the forked environment. Further details on how these variables are accessed can be found in Section 11.2.2.

Step 2) $\mathbf{T_{replay}}$ The second test method upgrades the forked environment so that USC(P, L_n) now uses L_{n+1} as its implementation. Typically, a smart contract upgrade involves deploying the new logic contract on the network and updating P to reference it. However, this procedure can be complicated by voting protocols and by differences in the specific upgrade pattern used. For example, the Transparent Proxy Pattern requires an external admin contract to execute an upgrade function, whereas other standards (e.g., the UUPS) do not have such a requirement. Rather than relying on upgrade-specific execution paths, Catana performs an "in-place" upgrade by directly replacing the bytecode at L_n 's address with that of L_{n+1}. Once this upgrade is performed, T_{replay} replays the same transaction t on the upgraded contract, extracting the new output and the new state snapshot. These will be compared against the baselines obtained by T_{oracle}. Note that if a transaction that succeeded on the original contract fails on the upgraded contract, the nature of the failure is taken into account. Unexpected failures (e.g., network time-outs) interrupt replay testing and are logged as such. Instead, a transaction revert is considered an symptom of a behavioral difference and is logged as part of a successful replay session.

Step 3) $\mathbf{T_{outcome}}$ The third test method compares the output-based and storage-based oracles with the actual outcomes captured by T_{replay}. A replay test session is deemed successful only if there are no deviations in output. However, Catana also registers and

logs any change in storage so that the tester can examine the impact of the upgrade more in-depth. Specifically, for each transaction replayed, it builds a log entry with the following information:

- **Transaction Data**: Information about the replayed transaction (e.g., its decoded arguments).
- **Output Changes**: A comparison of the transaction's output value on USC(P, L_n) and USC(P, L_{n+1}). The output value can be equal to the return value of the transaction (if any) or to an error message in case of revert.
- **Storage Changes**: A list of changes in the contract's storage variables after replaying the transaction on USC(P, L_n) and USC(P, L_{n+1}). Each change entry includes:
  - *Variable Data*: Key information about the storage variable (e.g., its name and type);
  - *Variable Value*: The value of the variable after replaying the transaction on L_n and L_{n+1};
  - *Storage Slot*: The storage slot of the variable after replaying the transaction on L_n and L_{n+1};
  - *Change Type*: The nature of the change. This can be a variable value change, a variable addition, a variable deletion or a storage slot change.
- **Status Code**: Summarizes whether the replay testing completed successfully (with or without logged changes), or resulted in error (either on the original or the upgraded contract).

Illustrative Example To better demonstrate how Catana operates, we present a faulty upgrade scenario involving the `CToken`[1] smart contract from the Compound Finance lending protocol. In this example (see Listing 11.1), when a user invokes the `mint` function of the `CErc20` contract (line 30), it calls `mintInternal` (line 23), which in turn calls `mintFresh` (line 3). The `mintFresh` function is responsible for minting new tokens and updating the `totalSupply` state variable, which tracks the total number of tokens in circulation. The original contract version correctly increments `totalSupply` by the `mintTokens` amount (line 14). However, the mutated version includes a bug (line 15) to simulate a faulty upgrade. The injected mutation subtracts 1 from the new `totalSupply` value, causing an underreporting in the total amount of tokens.

Listing 11.2 shows an excerpt of replay testing log produced by Catana when evaluating the faulty contract. When Catana replays a transaction that targets the `mint` method (line 2), it records both the changes in the transaction output (line 7) and in the contract's storage (line 11). Catana was able to detect the mutant in the `CToken` smart contract, although the issue was not immediately noticeable from the transaction's output. In fact, the return statement of the `mint` function (line 32 of Listing 11.1) represents a high-level success indicator. However, the mutation neither caused a transaction revert nor propagated to the return value of `mint`, which is still equal to 0 (indicating no error)

[1] CToken address: 0x3363BAe2Fc44dA742Df13CD3ee94b6bB868ea376

```
1  abstract contract CToken {
2
3       function mintFresh(address minter, uint mintAmount) internal {
4         uint allowed = comptroller.mintAllowed(address(this), minter,
              ↪ mintAmount);
5
6         if (allowed != 0) { revert MintComptrollerRejection(allowed);   }
7         if (accrualBlockNumber != getBlockNumber()) { revert
              ↪ MintFreshnessCheck(); }
8
9         Exp memory exchangeRate = Exp({mantissa:
              ↪ exchangeRateStoredInternal()});
10        uint actualMintAmount = doTransferIn(minter, mintAmount);
11
12        uint mintTokens = div_(actualMintAmount, exchangeRate);
13
14 --       totalSupply = totalSupply + mintTokens; // Original Version
15 ++       totalSupply = totalSupply + mintTokens - 1;  // Mutated Version
16
17        accountTokens[minter] = accountTokens[minter] + mintTokens;
18
19        emit Mint(minter, actualMintAmount, mintTokens);
20        emit Transfer(address(this), minter, mintTokens);
21    }
22
23     function mintInternal(uint mintAmount) internal nonReentrant {
24        accrueInterest();
25        mintFresh(msg.sender, mintAmount);
26    }
27 }
28
29 contract CErc20 is CToken {
30      function mint(uint mintAmount) override external returns (uint) {
31         mintInternal(mintAmount);
32         return NO_ERROR;
33      }
34 }
```

Listing 11.1: A possible Faulty Upgrade for the CToken Smart Contract

for both the original and the mutated contract (line 7 of Listing 11.2). Still, variables like `totalSupply` are critical to the contract's functionality, and any inaccuracy in its calculation can propagate and affect functions that rely on its value. As shown in the replay testing log, the `storageChanges` section (line 11) uncovers the discrepancy in the `totalSupply` value before and after the upgrade. This helps to visualize the effects of code changes that might not produce immediately obvious symptoms.

```
1  {
2     "tx": {
3       "hash": "0x14e6...d2b0",
4       "functionName": "mint(uint256_mintedAmount)",
5       "input": [{"mintAmount": "181380000000000000000"}]
6     },
7     "outputChanges": {
8       "valueBefore": "0",
9       "valueAfter": "0"
10    },
11    "storageChanges": [
12      {
13        "changeType": "value-changed",
14        "contract": "CErc20Delegate",
15        "name": "totalSupply",
16        "type": "t_uint256",
17        "slotBefore": "13",
18        "slotAfter": "13",
19        "decodedValueBefore": 4509423182276372383,
20        "decodedValueAfter": 4509423182276372382
21      }
22    ]
23  }
24 }
```

Listing 11.2: Excerpt of Catana log for the faulty CToken Smart Contract

11.2.2 Storage Analysis Workflow

During replay testing, Catana analyzes the storage of a Proxy contract to uncover changes that may surface when the connected Logic contract is upgraded. To understand how state variables are extracted and analyzed, we first refer to Ethereum's *storage model*, introduced in Section 2.3.3. Under this model, each smart contract maintains its storage as a *slot-value* database of state variables. Whenever Catana replays a transaction, it must retrieve each variable's data from its corresponding *slot* to reconstruct a complete snapshot of the Proxy's state. The Solidity compiler and existing testing utilities (e.g., `hardhat-storage-layout`) produce a Storage Layout describing the starting slot of each variable in storage. However, this layout provides an incomplete mapping for complex, nested, or dynamic types (e.g., arrays and structs), whose data can span multiple storage slots. If a replay test involves storage modifications to such data, eventual changes before and after the upgrade would go unnoticed. Catana addresses this limitation by building a **Complete Storage Layout (CSL)**, which maps every component of each variable to its specific location in storage. This richer view of the contract's internal state improves observability during testing and allows developers to know exactly where

changes occur after an upgrade. The storage analysis workflow implemented by Catana is depicted in Figure 11.2, and in the following we describe each step in detail.

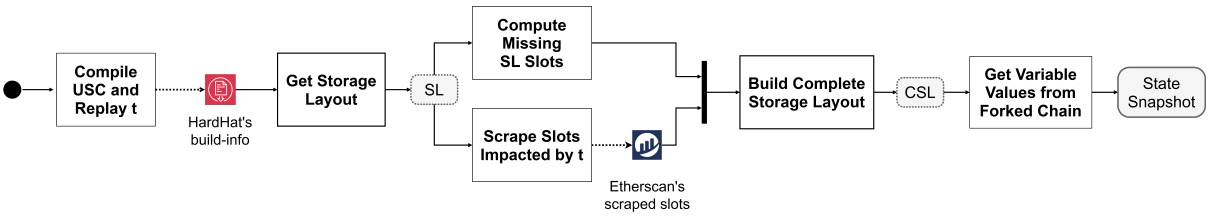


Figure 11.2: Storage Analysis Workflow implemented by Catana

1) Compile USC and Replay t As part of its replay testing workflow, Catana replays a transaction t on the the target USC (P, $\mathsf{L_n}$) so that its effects are committed to the Proxy's storage. As explained in Section 11.2, this process happens twice, on the original and on the upgraded versions of the USC.

2) Get Storage Layout After compilation, Catana extracts the **Storage Layout (SL)** of the target USC (P, $\mathsf{L_n}$) from HardHat's `build-info` files. The SL describes how the state variables of the smart contract are laid out in long-term memory. Each variable's "identifier" consists of two components: the *starting slot* `p`, indicating where the variable resides, and the byte *offset*, indicating where the variable starts within that slot.

3) Compute Missing Slots While the storage layout provides the starting storage slot `p` for each state variable, this information alone is insufficient for extracting the value of certain variable types. Dynamic, complex, or long data types, such as *arrays*, *structs* and long *bytes*, require additional steps to locate where their data is stored. For example, let us consider the dynamic array shown in Figure 11.3. The position of the array in storage is slot `p` = 508, however, the data at this location only describes the array's length. The actual array data starts at `keccak256(p)` and is laid out one element after the other. To manage this and other exceptions, Catana extends the initial SL by systematically computing the storage slots where the data of each state variable can be found, following the rules specified in the Solidity documentation.

4) Scrape Slots Impacted by t An additional consideration must be made for the *mapping* data type. Mappings in Solidity are used to store data in the form of key-value pairs and, unlike other data types, knowing the initial slot `p` of the mapping variable is not enough to locate its elements. In fact, as per the Solidity documentation, the value of a specific mapping element is found by applying a hashing function to its key. However, Solidity does not natively store iterable keys due to efficiency constraints in its resource-limited environment. So, if a contract does not explicitly provide the list of mapping keys (e.g., by storing them in a separate array), it is impossible to know them a-priori. One possible solution is to use the HardHat's `debug_traceTransaction`[2]. This method

[2]HardHat's debug_traceTransaction: `https://hardhat.org/hardhat-network/docs/overview#the--debug_tracetransaction--method`

returns a detailed trace of a mined transaction, providing a step-by-step breakdown of each executed opcode and the corresponding state of the EVM. By analyzing the opcodes, it is possible to identify the keys of mapping values affected by any t to be replayed. The drawback is that retrieving a single trace can take several minutes and relying on this method would make each replay test extremely inefficient. Instead, we implemented a web scraper that, given a transaction to be replayed t, extracts all the storage slots that were historically impacted by its execution. This information can be retrieved from *Etherscan*'s state change page[3] for any given transaction.

5) Build Complete Storage Layout Since Etherscan only provides raw storage slots, without the associated variable names or types, Catana identifies those belonging to mappings by checking if they are already present in the previously computed ones. After this step, the Complete Storage Layout (CSL) is ready.

6) Get Variable Values from Forked Chain After computing the CSL, Catana extracts the value of each variable from storage. The values are retrieved by calling `getStorageAt(address, slot)`[4] which, given the proxy address and a storage slot, extracts the respective value from the active fork. Once the CSL is expanded with the variable's values, Catana decodes them using the respective variable type so that they are in a human-readable format. An example of CSL with variable values for a dynamic array is shown in Figure 11.3. The output of this process permits Catana to compare the state snapshot for any two versions of an USC (P, $\mathsf{L_n}$) as described in Section 11.2.1.

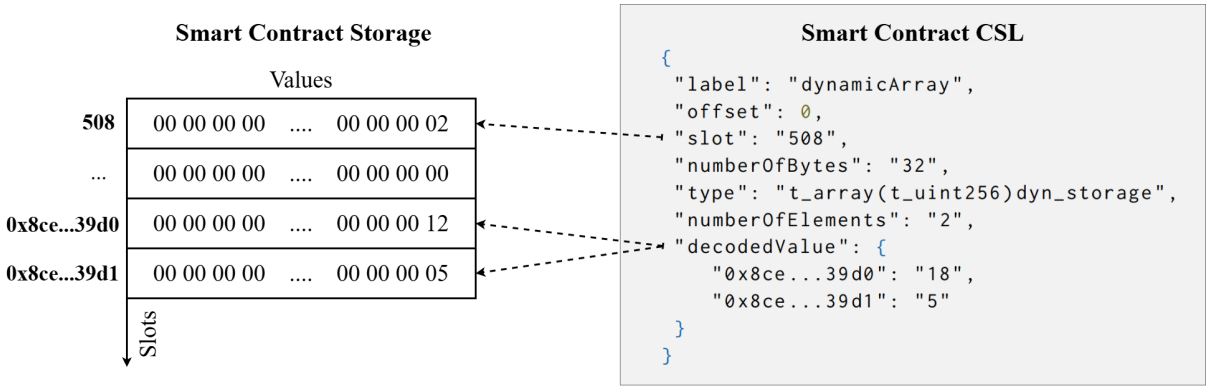


Figure 11.3: Example of Complete Storage Layout with Decoded Variable Values

11.3 Research Questions

In the following, we report the key research questions (RQs) that motivate this thesis. For each of them we briefly report the intent of the question and the strategy we planned

[3]Etherscan's state changes page for a transaction: `https://etherscan.io/tx/0x6ac79eda39a2d00ab9b4d3440c4672aa0493f99f89f265d8734b3a8aa9f28d14#statechange`

[4]HardHat's getStorageAt: `https://hardhat.org/hardhat-network-helpers/docs/reference#getstorageat(address,-index,-[block])`

to conduct the experimental studies.

RQ1: How does storage data analysis impact the effectiveness of Capture-Replay testing for USCs? RQ1 investigate to what extent incorporating smart contract storage analysis enhances Catana 's ability to detect disruptive upgrades. Smart contracts rely heavily on state variables for their operations, and an upgrade may alter how the contract interacts with its storage, potentially affecting the outcomes of future transactions. By analyzing state changes before and after an upgrade, we can gain additional insights into how modifications might influence the behavior of the USC. This is especially important in cases where subtle changes do not immediately manifest through transaction outputs (for example, through a revert). To answer this question, we simulate disruptive upgrades by generating mutants of USCs deployed on the Ethereum Mainnet. We then evaluate mutant detection by running a comprehensive replay test suite built from the USC's full transaction history. To measure the impact of storage analysis, we check how many faulty upgrades can exclusively be detected through changes in smart contract storage.

RQ2: Which transaction selection policy most effectively identifies disruptive behaviors in USCs? Replay testing is a powerful technique, but using the entire transaction history in real-world scenarios can be prohibitively expensive. Executing a replay test on a USC can take a few seconds due to overhead such as retrieving past blockchain states and configuring the local environment. Although this cost is acceptable when testing a single USC, it is still important to select transactions in a way that balances high fault-detection effectiveness against efficiency. Therefore, RQ2 investigates four distinct policies (*random*, *last*, *frequency*, and *unique*) for replay test selection. The study refers to several iterations, each one increasing the maximum size of the test suite (i.e., the maximum number of transactions to be replayed). We gauge each policy's performance primarily by the *Mutation Score*, which indicates how effectively a replay test suite identifies potentially disruptive changes. However, we also track the number of transaction replays needed to kill the injected mutants so as to measure testing overhead.

11.4 Experiment Methodology and Setup

In order to answer the research questions presented in Section 11.3, we need a set of Ethereum USCs that were upgraded since deployment. Although the proxy pattern is often embedded in real-world applications, developers approach upgrades with caution due to the connected risks and overall lack of supporting frameworks for regression testing. Thus, in practice, the underlying logic is replaced only infrequently [142, 101]. Since each USC typically features one or two past implementation versions, conducting large-scale experiments would require retrieving and testing hundreds of projects to capture a sufficient number of upgrade instances. To address this limitation, we simulate disruptive upgrades by generating mutants from the logic contract L of selected USCs. The rest of this section is organized as follow: Section 11.4.1 describes the experimental subjects, including the selection criteria for USCs and their characteristics. Sections 11.4.2 and

11.4.3 present the methodology and the evaluation metrics for RQ1 and RQ2. Lastly, Section 11.4.4 describes the hardware setup used for the experiment.

11.4.1 Subjects

To select the experimental subjects, we defined the following criteria:

1. The subject must be a proxy-based USC deployed on the Ethereum Mainnet. This allows us to capture transactions from real end-users.

2. The source code of the USC must be verified on Etherscan. Access to the ABI and source code is essential for interacting with the contracts during testing and for extracting their storage layout.

3. The USC must be written in Solidity and use version 0.5.13 or later. This criterion is set because prior versions of the Solidity[5] compiler did not produce the `storageLayout` output.

4. The USC must feature up to 3,000 successfully executed transactions. To capture all relevant user behaviors, we plan to use the complete transaction history of a USC as a replay test suite. Setting this upper bound keeps the experiment computationally manageable. In fact, one replay test requires approximately 10 seconds[6] to run, and executing thousands of transactions on hundreds of mutants can require many days of computation per project.

5. The USC must feature more than 500 successfully executed transactions. Projects with fewer transactions may not provide enough data to draw reliable conclusions.

To find viable projects, we used the smart contract search feature provided by Etherscan[7]. We used the "*Proxy*" keyword and sorted the results by latest to prioritize recent Solidity versions (the query was conducted in July 2024). Among the collected results, we selected 5 projects respecting all criteria. Although this final selection is limited, each USC is mutated to simulate numerous faulty upgrades (more details about this are provided in Section 11.4.2). This design keeps the experiment's total runtime manageable while still providing comprehensive insight into the capabilities of replay testing.

Table 11.1 lists the subjects selected for the experiment, including the address of the proxy P, the address of its current logic contract L, and the total number of transactions $\#T$ that were successfully executed on it at the time of writing (i.e., until $\mathit{EndBlock} = 20357000$). If the subject had never been upgraded since deployment, we extracted all the transactions executed on P from block 0 to *EndBlock*. Otherwise, we identified the block b including the upgrade transaction for L, and we set $\mathit{StartBlock} = b + 1$. This ensures that all the extracted transactions target the latest logic contract version L. Regarding the transaction history, Table 11.1 also shows the total number of unique methods u that were invoked at least once, out of all methods belonging to the USC's interface (i.e., its

[5]Solidity 0.5.13 release announcement: `https://soliditylang.org/blog/2019/11/14/solidity-0.5.13-release-announcement/`

[6]Based on prior empirical experience

[7]Etherscan's smart contract search feature: `https://etherscan.io/searchcontractlist`

Table 11.1: Experimental Subjects

ID	Name	Contract Addresses	StartBlock	EndBlock	#T	u/#ABI
1	BSTPStaking	P: 0x57...c73e L: 0x5d...9088	16483398	20357000	1 175	4/18
2	Paladin	P: 0x24...53e5 L: 0xcf...bb2b	14880569	20357000	578	3/20
3	GMMToken	P: 0x4B...c279 L: 0x60...Ce2F	0	20357000	2 547	4/17
4	Lucids	P: 0x7D...8AD0 L: 0xea...5aA6	0	20357000	1 526	10/29
5	DeDudes	P: 0xC2...91DD L: 0x8d...dadc	0	20357000	2 368	13 /31

ABI). Note that we excluded *pure* and *view* functions from the $u/\#ABI$ ratio, even if these are technically part of a smart contract's interface. Indeed, these are read-only functions that cannot change the state of the smart contract, meaning their execution does not produce recorded transactions.

11.4.2 Methodology for RQ1

RQ1 examines the impact of smart contract storage analysis on the effectiveness of capture-replay testing. To answer this question in a comprehensive way, we observed how disruptive upgrades (simulated through mutants) are detected when using the full transaction history as a replay test suite. Below we detail the steps taken to answer RQ1:

1. **Replay Test Selection:** For each USC (P, L), we retrieved the set of all successfully executed transactions T (see Table 11.1) recorded on the Ethereum Mainnet. We use these to build a *full* replay test suite, setting an upper bound of possible interactions for disclosing mutants.

2. **Mutant Generation:** We generated a set of mutants for the Logic contract L, excluding interfaces, utilities, and linked libraries. Each mutant introduces a single fault into the original contract code. We generated the mutants using SuMo [22], a mutation testing tool for the Solidity language. Specifically, we applied all the mutation operators (both traditional and Solidity-specific) presented in the most recent literature [10]. After mutant generation, we applied random sampling to select a manageable subset of 100 mutants.

3. **Replay Testing:** We ran Catana on each (compilable) mutant $m \in M$ using the *full* replay test suite. This process produces a *replay matrix* detailing the outcome of each transaction $t \in T$ on each mutant m.

4. **Killed Mutants Analysis:** From the replay matrix, we identified the set of *killed* mutants $K_{\text{full}} \subseteq M$. A mutant m is considered *killed* if at least one transaction in the *full* replay test suite could detect it using either of the following killing conditions:

- **Output Killing Condition:** There exists at least one transaction $t \in T$ for which the output produced by the original contract differs from that produced by mutant m. This includes any discrepancy in the return value or in the revert behavior of the transaction (e.g., the transaction is successfully executed on the original contract but it reverts on the mutant).
- **Storage Killing Condition:** There exists at least one transaction $t \in T$ that results in a change in the contract's storage state when executed on the original contract compared to mutant m. This includes scenarios where at least one state variable in m is added, deleted, altered in value, or moved to a different storage slot.

To answer RQ1, we examine the outcome of Catana on the killed mutants K_{full}, categorizing them as follows:

- $K_o \subseteq K_{\text{full}}$: is the subset of mutants killed exclusively by the **output killing condition**, indicating detectable functional differences in the contract's response[8];
- $K_s \subseteq K_{\text{full}}$: is the subset of mutants killed exclusively by the **storage killing condition**[9];
- $K_{os} \subseteq K_{\text{full}}$: is the subset of mutants killed by both the **output** and the **storage killing condition**.

To consider the storage-based replay testing approach as effective, the mutants in K_s should represent a meaningful contribution to K_{full}. A significant K_s implies that storage data extraction can enhance Capture-Replay testing for USCs, uncovering unique issues that output-based conditions might miss.

11.4.3 Methodology for RQ2

RQ2 explores different policies for selecting transactions from the complete history and evaluates how the generated test suites perform in detecting disruptive upgrades. To this end, we compared their effectiveness and efficiency in killing USC mutants. Below we detail the steps taken to answer RQ2:

1. **Replay Test Selection:** To build the replay tests suites to be evaluated, we applied several selection policies to the full transaction history T of each target USC (P, L). The selection policies are:
   - **Random:** This policy randomly selects n_R transactions from T, providing broad coverage without bias towards specific blocks or methods.
   - **Last:** This policy selects the last n_L transactions in T based on their block number, focusings on recent user behaviors that may reflect the latest usage patterns.

[8] in Mutation Testing these mutants are said killed under Strong Mutations [111]
[9] in Mutation Testing these mutants are said killed under Weak Mutations [111]

- **Frequency:** This policy selects n_F transactions so that the sample mirrors the original method distribution in T. This guarantees that the resulting test suite includes transactions for those methods that are most frequently invoked in real-world usage.
- **Unique:** This policy partitions n_U across the distinct methods in T, sampling an equal number of random transactions for each method. This ensures that every method is represented at least once in the final sample, maximizing functional diversity.

For each project, n determines the maximum **size of the replay test suite** built using the different policies (i.e, $n = n_R = n_L = n_F = n_U$). This is computed as $n = u \times i$, where u represents the total number of unique method signatures in the transaction history (as shown in Table 11.1), and i is the scaling factor. In the experiment, we incrementally increased the value for i to observe the effect of different test suite sizes on mutant detection.

2. **Extraction of Killable Mutants:** Because some contract statements in L may not be invoked by any transaction in T, certain mutants in M might be inherently undetectable. Such mutants would not provide meaningful insight to this comparative study, so the set of *killable* mutants for RQ2 is given by $K_{\text{full}} \subseteq M$. Indeed, these are the mutants that were *actually* killed when using the full transaction history in RQ1 as a replay test suite (regardless of the considered killing condition).

3. **Replay Testing:** To evaluate each policy's effectiveness, we run the generated replay test suites against the mutants in K_{full}. Since we must run replay testing for varying test suite sizes and multiple repetitions, this process is computationally expensive. For each policy, we repeat the selection and the replay testing process 10 times. The only exception is the policy *last* which is fully deterministic and not influenced by randomness.

To answer RQ2, we compare the defined transaction selection policies considering the following metrics:

1. MS: The **Mutation Score** (MS) represents the proportion of killed mutants $K \subseteq K_{\text{full}}$ detected by a replay test suite. A higher MS estimates better effectiveness in detecting potential issues in the USC Under Test.

2. *Replays*: The number of **transaction replays** (i.e., test executions) necessary to achieve a certain MS value. This metric represents the efficiency of the test suite in terms of transaction usage. Fewer replayed transactions indicate a more efficient and targeted approach.

11.4.4 Hardware Setup

We conducted the experiments by running Catana on Ubuntu Linux Virtual Machines with 16 GB RAM and 1vCPU. For the extraction of historical blockchain data, Catana supports forking of the main network through Hardhat. To use this feature, Hardhat must connect to an archive node, which is an instance of an Ethereum client configured to hold

historical blockchain states. For typical usage of Catana having a personal archive node is not necessary, as users can rely on free services such as Alchemy or Infura. However, access through these services is mostly limited and not suitable for intensive use. Since our experiments involve replaying thousands of transaction executions on numerous mutants, we set up our own archive node using Erigon [56], an execution layer client designed to offer an optimized implementation of Ethereum. This allows us to handle the extensive data requirements of our experiments while maintaining consistent performance.

11.5 Discussion of Results

In this Section, we discuss the results of the experiment and answer the research questions posed in Section 11.3. The data supporting the findings of this study are publicly available on GitHub [119]. We recall that the objective of RQ1 is to evaluate whether storage analysis can significantly improve Catana's ability to detect disruptive smart contract upgrades. Instead, RQ2 studies the effectiveness of several transaction selection policies in identifying any change exhibited by upgraded USCs (i.e., in the output, in the storage, or in both). To address both these questions, we first used the full transaction history T of each selected USC (P, L) to run replay testing on mutants of their logic contract L. Table 11.2 summarizes the 500 randomly generated mutants across all projects. It outlines the corresponding mutation operators and provides a code snippet to illustrate how each one alters Solidity contracts. A single mutation operator may embed multiple mutation rules, thus the example is intended for illustrative purposes only. For a more detailed explanation of the operators, please refer to Chapters 6 and 8.

Table 11.3 summarizes the replay testing results for each subject. Here, $\#M$ indicates the total number of mutants that could be compiled and tested, while $\#T$ is the size of the replay test suite (i.e., the total number of successful transactions available on Ethereum). The total number of replay test executions for each project is computed as $\#(M \times T)$. Although generating a complete replay matrix can be time intensive (testing continues even after a mutant is detected early, resulting in more than 800k replays), this process yields a full *mutant-transaction* mapping that we use for subsequent analyses. Within M, the subset K_{full} denotes those mutants that were killed by at least one transaction in T (regardless of which killing condition triggered detection), while L contains the live mutants that remained undetected. The Mutation Score (MS) for the *full* replay test suite T indicates the upper bound on potential mutant detection for any smaller test selection performed on T.

In Section 11.5.1, we focus on the subset of mutants that the full transaction history T was able to kill and analyze the specific factors contributing to their detection. In Section 11.5.2, we present a comparative study of different transaction selection policies applied to T, examining both their effectiveness and efficiency in detecting the killable mutants K_{full}. Based on this investigation, Section 11.5.3 outlines insights and guidelines for practitioners aiming to ensure the reliability of their contract upgrades.

Table 11.2: List of Mutants randomly sampled from those generated by SuMo's Mutation Operators

Mutation Operator	**Illustrative example of a possible mutation**	**Mutants**
ACM - Arg. Change of overloaded Method call	`foo(a,b) → foo(a,b,c)`	4
AMR - Array Member Replacement	`a.push(x) → //a.push(x)`	1
AOR - Assignment Operator Replacement	`balances[to] += 1 → balances[to] = 1`	5
AVR - Address Value Replacement	`foo(address_a) → foo(address_b)`	33
BOR - Binary Operator Replacement	`a + b → a - b`	169
BVR - Boolean Value Replacement	`true → false`	20
CER - Casting Expression Replacement	`uint64(a) → uint56(a)`	3
CSC - Conditional Statement Change	`if(condition) → if(true)`	30
EHD - Exception Handling Deletion	`revert() → //revert()`	30
FCD - Function Call Deletion	`foo() → //foo()`	16
GVR - Global Variable Replacement	`msg.value → msg.value-1`	5
ISD - Initialization Statement Deletion	`initialize(){ setFee(1); } → initialize(){}`	4
LSC - Loop Statement Change	`while(condition) → while(false)`	2
PLR - Precision Loss Replacement	`(a * b) / c → (a / c) * b`	8
MCR - Math-Crypto function Replacement	`keccak256 → sha256`	10
MOD - Modifier Deletion	`function foo() onlyOwner {} → function foo() {}`	25
NVR - Number Value Replacement	`1000 → 1001`	95
OLFD - Overloaded Function Deletion	`function foo(){} → /*function foo(){}*/`	10
ORFD - Overridden Function Deletion	`function foo() override {} → /* function foo() override {}*/`	11
RSD - Return Statement Deletion	`return fee → //return fee`	10
RVS - Return Values Swap	`foo() returns (uint a, uint b) → foo() returns (uint b, uint a)`	1
SKID - Super Keyword Insertion and Deletion	`supply → super.supply`	3
SVR - String Value Replacement	`"TOKEN" → ""`	1
TRC - Transfer Return Check	`require(transferTo(to, amount))) → transferTo(to, amount)`	2
UORD - Unary Operator Replacement Deletion	`!a → a`	2
Total		**500**

Table 11.3: Results of Replay Testing using the Transaction History T as a Test Suite

ID	$\#T$	$\#M$	$\#K$	$\#L$	MS	*Replays*	*Time*
1	1 175	98	48	50	48.9%	117 500	205 h
2	578	95	20	75	21%	57 800	85 h
3	2 547	92	12	80	13%	254 700	509 h
4	1 526	93	73	20	78.5%	152 600	403 h
5	2 368	90	85	5	94.4%	236 800	596 h
Tot.	8 194	468	238	230	50.9%	819 400	1 798 h

Legend:

#T:	Transactions in the Test Suite
#M:	Tested Mutants
#K:	Killed Mutants
#L:	Live Mutants
MS:	Mutation Score = $(\#K/\#M) * 100$
Replays:	Number of Replay Test Executions = $\#(T \times M)$

11.5.1 Answering RQ1 - Impact of Storage Analysis

With RQ1, we investigate whether and to what extent the extraction of storage data influences Capture-Replay testing outcomes in Upgradeable Smart Contracts. To answer this question, we focus on the subset of mutants K_{full} that were killed when replaying the *full* transaction history T (see Table 11.3), exploring the reason for their detection. Table 11.4 further categorizes these mutants by killing condition into $\#K_o$ (mutants detected by a change in *output*), $\#K_s$ (mutants detected by a change in *storage*), and $\#K_{os}$ (mutants detected by *both*).

Table 11.4: Mutants Killed by the full Transaction History T

ID	$\#M$	$\#K_{\textbf{full}}$	$\#K_o$	$\#K_s$	$\#K_{\text{os}}$	MS
1	98	48	0	42	6	48.9%
2	95	20	2	7	11	21%
3	92	12	2	6	4	13%
4	93	73	0	70	3	78.5 %
5	90	85	0	81	4	94.4%
Tot.	468	238	4	206	28	50.9%

Legend:

#M:	Tested Mutants
$\#K_{\text{full}}$:	Mutants Killed by T
$\#K_{\text{o}}$:	Mutants Killed by an Output Change
$\#K_{\text{s}}$:	Mutants Killed by a Storage Change
$\#K_{\text{os}}$:	Mutants Killed by Output and Storage Changes

As shown in Figure 11.4, a replay testing approach that relies solely on examining transaction outputs is highly limited. In fact, only around 13.5% ($K_o \cup K_{\text{os}}$) of the killable mutants across all projects produced observable output differences with respect to the original smart contract. Specifically, 11.8% of the mutants (K_{os}) resulted both in output and storage changes during replay testing. Within this small set of 28 mutants, all of them were detected because a replayed transaction passed on the original contract but reverted on the mutated one. Since the revert prevented the expected storage modifications from taking place, Catana also logged changes in the contract's state variables. Instead, 1.7% of the mutants (K_o) were detected by observing exclusively the output of a transaction. Of these 4 mutants, one of them was killed by a revert, and the remaining three due to changes in the transaction's return value. The majority of the killable mutants (86.5%) could only be detected by observing changes in the smart contract's storage.

In general, smart contract methods that can potentially change the state (i.e., those not marked as *pure* or *view* in Solidity) do not return values directly to the external caller.

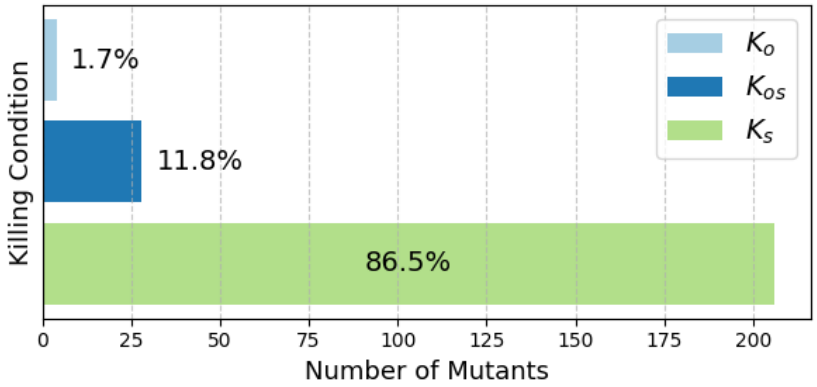


Figure 11.4: Distribution of Mutants in K_{full} by Killing Condition

While it is technically possible to simulate the execution of such methods and observe their outputs (e.g., via a static call), the return values from the contract's internal execution are often lost or sometimes simplified at the ABI (Application Binary Interface) level. For example, state-changing functions in the ABI might return higher-level indicators, such as whether the transaction succeeded or failed (as in the illustrative example in Listing 11.1), rather than more granular data. Therefore, unless a code change causes a transaction to revert, its effects frequently remain hidden within the contract's state variables and are not immediately evident.

Table 11.5: Types of Changes that triggered Mutant detection

Set	**Mutants**	**Tx. Output Changes**		**State Variable Changes**			**Total**
		Return Value	*Revert*	*Value*	*Slot*	*Value and Slot*	
K_o	4	3	1	-	-	-	4
K_s	206	-	-	242	3	5	250
K_{os}	28	0	28	63	0	0	91

Legend:
K_o: set of mutants exclusively killed by output changes
K_s: set of mutants exclusively killed by storage changes
K_{os}: set of mutants killed by both output and storage changes

As detailed in Table 11.5, all mutants in K_s collectively caused 250 changes in state variables: 242 of these involved value changes, 3 involved slot changes, and 5 involved both slot and value changes. There is also a notable imbalance in the types of variables typically affected. The vast majority of changed values are elements of a mapping (82%), followed by unsigned integers (14.4%), and a small percentage of address (1.6%), boolean (1.2%), and string (0.8%) data types. Although the distribution of affected variables depends on the specifics of each project, mappings are clearly critical to smart contract functionality. In fact, they are frequently used to store user data, such as token balances, staking amounts, or configuration parameters. Thus, being able to locate and inspect their elements is crucial to fully understand the impact of code changes and prevent

potential issues in real-world contracts.

> **Answer to RQ1** *How does storage data analysis impact the effectiveness of Capture-Replay testing for USCs?*
>
> Integrating storage analysis in replay testing is essential for flagging upgrades that might disrupt the behavior of the USC, leading to regressions and compatibility concerns. In our experiment, 86.5% of the killable mutants could exclusively be detected by observing changes in the smart contract's state variables.

11.5.2 Answering RQ2 - Analysis of the Transaction Selection Policies

The results from RQ1 demonstrated that Capture-Replay testing can effectively detect disruptive changes in Upgradeable Smart Contracts (USCs). However, using the entire transaction history T as a replay test suite may be unfeasible, especially for projects with hundreds of thousands of historical transactions. Running a single replay test takes on average 7 seconds, mostly because Catana must fork the Ethereum Mainnet and retrieve historical states. While this overhead is still reasonable when testing a single USC (as opposed to a large number of mutants), it becomes crucial to select only the most relevant transactions in order to maximize effectiveness (i.e., fault-detection capabilities) and efficiency (i.e., reduced replay time).

Therefore, RQ2 investigates how different transaction selection policies (*random*, *last*, *frequency* and *unique*, as defined in Section 11.4.3) detect disruptive behaviors in USCs. This comparative study evaluates replay test suites of different sizes, progressively allocating a higher budget for the testing activities. We recall that the size n of a test suite is computed as $n = u \times i$, where u is the number of unique method signatures in a project's transaction history, and i is the scaling factor. For context, Table 11.6 shows the u value for each project, together with the size n of its replay test suite for the smallest ($i = 1$), medium ($i = 10$), and largest ($i = 20$) value of the scaling factor. For example, the BSTPStaking project (with 4 unique method signatures) yields a minimum test suite size of 4 transactions and a maximum of 80 transactions (4×20). The maximum number of replays for a given test suite is given by $n \times K_{\text{full}}$. This upper bound is reached in the worst-case scenario when no mutant is detected early.

The generated replay test suites are evaluated considering two key metrics. The primary evaluation metric is the **Mutation Score** (MS), as it measures how effectively a replay test suite uncovers changes within upgraded contracts. However, the study also evaluates the efficiency of each test suite by tracking how many **Replays** are needed to kill the injected mutants. Figure 11.5 illustrates the mutant-detection capabilities of *random*, *last*, *frequency* and *unique* for each tested project (averaged over 10 repetitions of the experiment). The y-axis shows the Mutation Score achieved by the generated test suite, while the x-axis represents the scaling factor i, which increases the test suite size linearly. In each iteration i, we retain all $u(i-1)$ transactions selected in the previous iteration and add u new transactions. Table 11.7 provides a concrete time benchmark for these policies considering the reference test suite sizes of Table 11.6. Each cell shows the total time (in *hours*) required for completing replay testing on the mutants in K_{full}, and

Table 11.6: Details about the Smallest, Medium, and Largest Test Suite per project

ID	$\#K_{full}$	u	Test Suite (for $i = 1$)		Test Suite (for $i = 10$)		Test Suite (for $i = 20$)	
			Size n	Max. Replays	Size n	Max. Replays	Size n	Max. Replays
1	48	4	4 tx	192	40 tx	1 920	80 tx	3 840
2	20	3	3 tx	60	30 tx	600	60 tx	1 200
3	12	4	4 tx	48	40 tx	480	80 tx	960
4	73	10	10 tx	730	100 tx	7 300	200 tx	14 600
5	85	13	13 tx	1 105	130 tx	11 050	260 tx	22 100

Legend:
$\#K_{full}$: number of killable mutants in the project
u: number of unique method signatures in the project's transaction history
i: scaling factor
n: replay test suite size (computed as $u \times i$)

the corresponding Mutation Score. For comparison, the last column refers to the test suite built from the full transaction history T.

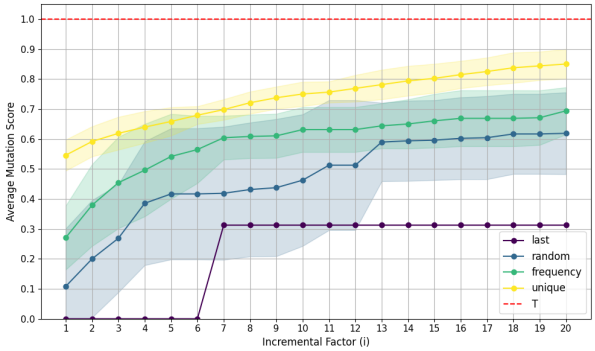
Instead, the average performance of each policy on all the considered projects is summarized in Figure 11.6. Here, the left y axis represents the effectiveness of a policy in terms of its average *Mutation Score*, while the right y axis indicates its efficiency in terms of average *Replays*. In the following, we discuss the results by policy in more detail.

Last Policy The *last* policy selects the most recent n transactions from the transaction history, focusing on the ABI methods that users have recently invoked. In this case, the test suite size n directly impacts the length of the temporal window from which the transactions are selected. However, since the content of this window is dependent on the behavior of users, the resulting test suite might lack diversity in the methods it covers. Therefore, it often struggles to provide comprehensive functional coverage and detect a broad range of mutants. From the charts in Figure 11.5, *last* consistently achieves the lowest Mutation Score of all policies. This gap is particularly evident for projects like BSTPStaking (Figure 11.5a), Paladin (Figure 11.5b), and GMMToken (Figure 11.5c), where users engage with only a limited number of methods in the ABI (resulting in very low u values). Figure 11.6 further shows that, at its smallest test suite (i = 1, $\overline{n}$ = 7 transactions), *last* achieves an average $\overline{MS}$ = 0.36 with 367 test executions, which is only slightly below the *random* policy. However, due to the strong dependence on user-specific behavior, *last* becomes increasingly ineffective and unpredictable as i grows. In fact, selecting redundant transactions inflates the overall number of test executions without proportionate gains in Mutation Score. For instance, at the largest test suite tested (i = 20, $\overline{n}$ = 136 transactions), *last* only reached $\overline{MS}$ = 0.6 at the cost of more than 5300 replay test executions. Although it is not especially effective as a stand-alone policy, *last* can be a valuable addition to other strategies by capturing newly emerging user behaviors that might otherwise be overlooked.

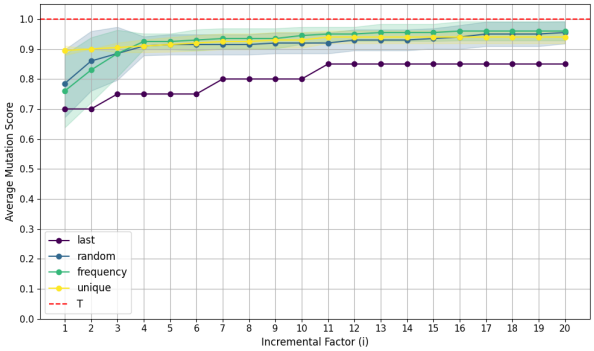
Random Policy The *random* policy serves as a baseline approach, offering broad sampling without focusing on specific methods or block numbers. As shown in Fig-

Table 11.7: Average Results of Replay Testing on the Mutants (Time in hours, and % of the Killed Mutants over K_{full})

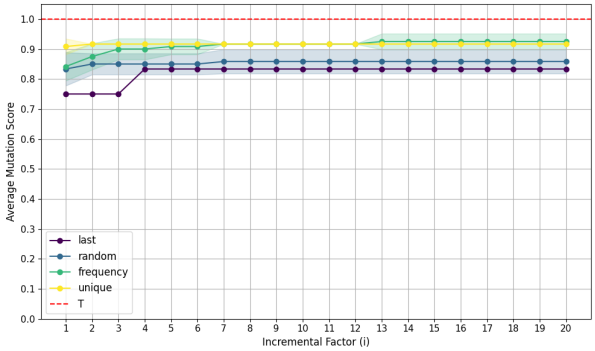
ID	Policies												T
	Last			Random			Frequency			Unique			
	i=1	**i=10**	**i=20**	**i=1**	**i=10**	**i=20**	**i=1**	**i=10**	**i=20**	**i=1**	**i=10**	**i=20**	Full History
1	**0.36h**	**3.23h**	**5.64h**	**0.34h**	**2.45h**	**4.00h**	**0.37h**	**2.01h**	**3.29h**	**0.29h**	**1.14h**	**1.69h**	**21.2h**
	0.0%	31.3%	31.3%	10.8%	46.3%	61.9%	27.0%	63.1%	69.4%	54.6%	75.0%	85.0%	100.0%
2	**0.06h**	**0.26h**	**0.40h**	**0.06h**	**0.14h**	**0.20h**	**0.05h**	**0.13h**	**0.17h**	**0.05h**	**0.12h**	**0.17h**	**0.8h**
	70.0%	80.0%	85.0%	78.5%	92.0%	95.5%	76.0%	94.5%	96.0%	89.5%	93.0%	94.0%	100.0%
3	**0.06h**	**0.23h**	**0.39h**	**0.04h**	**0.17h**	**0.31h**	**0.06h**	**0.15h**	**0.23h**	**0.05h**	**0.12h**	**0.17h**	**4.9h**
	75.0%	83.3%	83.3%	85.0%	85.8%	85.8%	87.5%	91.7%	92.5%	91.7%	91.7%	91.7%	100.0%
4	**2.13h**	**13.21h**	**23.10h**	**1.89h**	**15.02h**	**26.23h**	**1.91h**	**13.90h**	**23.43h**	**1.48h**	**5.46h**	**8.60h**	**58.6h**
	1.4%	50.7%	60.3%	12.5%	38.8%	51.2%	16.3%	46.3%	59.32%	31.8%	55.6%	67.0%	100.0%
5	**2.57h**	**17.97h**	**42.99h**	**2.98h**	**18.70h**	**37.76h**	**2.88h**	**15.93h**	**28.49h**	**2.34h**	**7.25s**	**10.94h**	**194h**
	35.3%	37.7%	41.2%	13.5%	43.4%	58.4%	21.9%	59.3%	75.8%	36.9%	67.4%	71.1%	100.0%



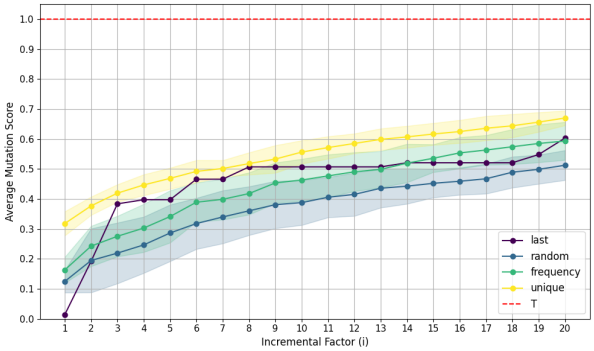
(a) Mutation Score for BSTPStaking over 48 Killable Mutants



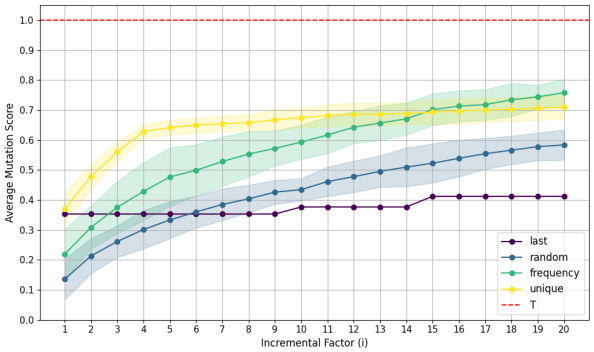
(b) Mutation Score for Paladin over 20 Killable Mutants



(c) Mutation Score for GMMToken over 12 Killable Mutants



(d) Mutation Score for Lucids over 73 Killable Mutants



(e) Mutation Score for DeDudes over 85 Killable Mutants

Figure 11.5: Average Mutation Score Achieved by Each Transaction Selection Policy for Increasing Test Budgets

ure 11.5, test suites with randomly selected transactions perform moderately well across all projects, but their performances vary significantly across multiple experiment repetitions. Allocating a larger test budget pays off in terms of mutant detection, with the Mutation Score steadily increasing as more transactions are incorporated into the replay test suite. Despite this gradual improvement, for any given value of i, ***random*** is always outperformed by more focused policies like ***frequency*** and ***unique***. When considering its average performances over all projects, Figure 11.6 shows that ***random*** achieves a $\overline{MS}$ = 0.4 at the lowest value of i (where $\overline{n}$ = 7 transactions). This places it just above the

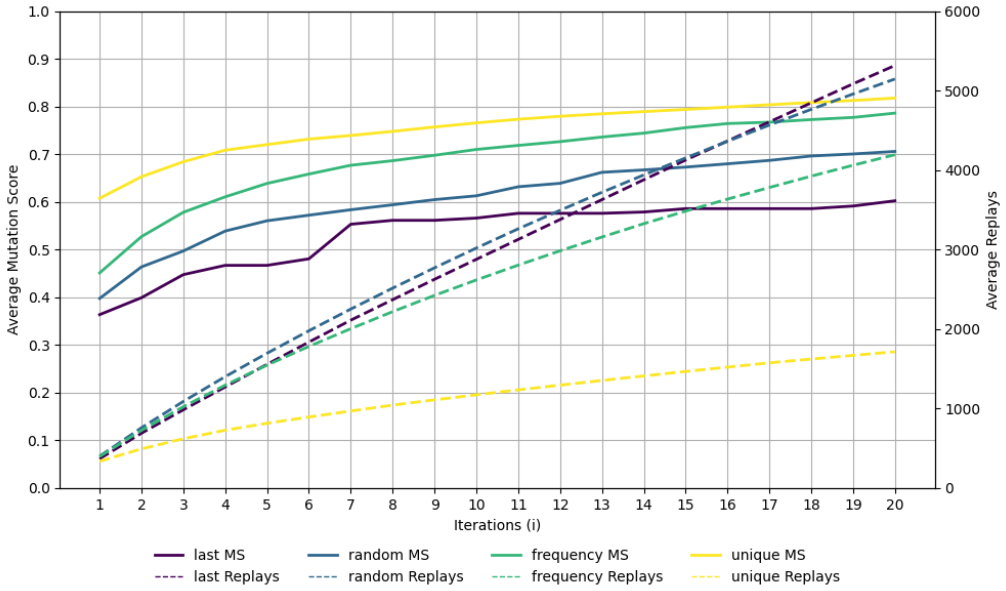
last policy, ($\overline{MS}$ = 0.36), but still below the *frequency* policy ($\overline{MS}$ = 0.45) for roughly the same number of replays. As the test suite size grows, this gap in fault-detection becomes more evident. With the largest test suite we evaluated ($\overline{n}$ = 136 transactions), *random* reaches a moderate $\overline{MS}$ = 0.71 with around 5100 test executions. With the same test suite size, *frequency* not only achieves a higher $\overline{MS}$ = 0.79, but also detects the injected changes faster, with ≈ 1000 fewer replays. Therefore, although *random* can eventually achieve adequate Mutation Score values, it tends to require more replays and remains inherently more unpredictable in its results.

Frequency Policy The *frequency*-based selection ensures that heavily used methods are proportionally well-represented in the test suite. This approach prioritizes testing core contract methods that are frequently invoked, but it may neglect less common functions in rarely explored code paths. Like for the *random* policy, it is evident that the selected transactions for small values of i may not extend beyond those commonly invoked by the users. Once the size of the test suite grows, *frequency* systematically broadens the diversity of test inputs for frequent transactions while also incorporating replay tests for less common methods. This tends to drive up Mutation Score faster and more reliably than random sampling alone. This trend is evident in Figure 11.6, where the average Mutation Score across projects grows from an initial 0.45 (for i = 1, $\overline{n}$ = 7 transactions) to almost 0.65 (for i = 5, $\overline{n}$ = 35 transactions) in the span of five iterations. Once i = 20, *frequency* achieves a maximum $\overline{MS}$ = 0.79 transactions, and displays the highest fault-detection capabilities on the Paladin (Figure 11.5b), GMMToken (Figure 11.5c) and DeDudes (Figure 11.5e) projects. Overall, *frequency* performs reasonably well even with modest test suite sizes. When balanced properly, it can offer strong fault-detection capabilities and cost-effectiveness, making it a suitable choice for most projects.

Unique Policy The *unique* policy evenly distributes transaction selections across all methods invoked in a contract's transaction history. Unlike *frequency*, it prioritizes test diversity over repeatedly targeting high-frequency transactions. This approach ensures that every method in the history is tested in at least one configuration, even at the lowest test budget. Remarkably, *unique* achieves relatively high Mutation Scores even with a minimal test suite consisting of one transaction per invoked method. For instance, in projects like BSTPStaking (Figure 11.5a), Paladin (Figure 11.5b) and GMMToken (Figure 11.5c), the Mutation Score reached by *unique* at its lowest test suite size (i = 1) surpasses that achieved by *last* at its highest (i = 20). When considering the average performance over all projects (Figure 11.6), *unique* outperforms every other policy at the lowest i, achieving $\overline{MS}$ = 0.6 against the $\overline{MS}$ = 0.45 of the closest competitor (i.e., *frequency*). Then, as the i factor increases, the test suite expands with transactions that explore a wider variety of initial contract states and parameters (i.e., test inputs). As a result, it achieves the highest Mutation Score of all policies on two out of five projects (BSTPStaking in Figure 11.5a, and Lucids in Figure 11.5d), and only slightly below *frequency* on the remaining three projects. When considering the largest test suite size (i = 20, $\overline{n}$ = 136 transactions), *unique* achieves the highest Mutation Score of all policies ($\overline{MS}$ = 0.82). Although this value is only slightly ahead of *frequency*'s ($\overline{MS}$ = 0.79), *unique* reaches it with approximately 60% fewer transaction replays. Indeed, *frequency*

may require a significantly higher number of replays due to over-representation of popular methods, increasing the likelihood of redundant tests. Additionally, *unique* displays a narrow confidence interval, indicating that it is not only effective but also stable and reliable, with minimal variability in mutant-detection performance.

Figure 11.6: Average Mutation Score and Transaction Replays achieved across Policies



Answer to RQ2 *Which transaction selection policy most effectively identifies disruptive behaviors in USCs?*

The *unique* policy, which focuses on method-level diversity, outperforms other strategies in terms of mutation adequacy and number of replayed transactions. In the experiment, we observed that even the minimal replay test suite achieves good fault-detection capabilities, killing 60% of detectable mutants.

11.5.3 Guidelines for Practitioners

Based on our investigation of Capture-Replay testing for Upgradeable Smart Contracts, we outline the following insights and guidelines for practitioners aiming to ensure the reliability of their contract upgrades.

First, our results in Section 11.5.1 show that focusing solely on transaction outputs detects a limited subset (≈ 13.5) of potentially harmful changes. Many smart contract functions, especially state-changing ones (i.e., neither pure nor view), do not return meaningful values at the ABI level. Therefore, a revert might be the only outward signal of a problematic upgrade. When using Capture-Replay testing, practitioners should **carefully examine internal state changes** (e.g., storage variables) and not rely exclusively on transaction outputs. Catana ensures that developers gain visibility into critical data structures, such as mappings that hold user balances and other key application data.

Instead of guessing where an upgrade might break the logic, developers can get precise feedback on which specific variables diverged from their expected values.

As shown in Section 11.5.2, adopting the right transaction selection policy is essential in balancing efficiency (minimizing time) and effectiveness (maintaining high fault detection). The optimal test selection depends on the project's needs and testing budget (i.e., how many transactions should be included in the replay test suite). Our findings suggest that small, method-diverse replay test suites can still provide fast feedback with reasonable effectiveness. Employing specific policies allows Catana to support both quick smoke tests (*e.g.*, ***unique*** with small test suites) and thorough regression testing (*e.g.*, ***frequency*** or ***unique*** with larger budgets) for high-stakes upgrades.

For example, when using the ***unique*** policy on only one transaction per invoked method (as retrieved from the project's on-chain history), Catana can quickly verify whether the upgraded contract's core functionality remains intact. Because ***unique*** ensures that every method is tested at least once, even the smallest test suite can detect disruptive changes in rarely executed paths without devoting extensive time to testing. This approach enables a lightweight validation, with a test suite completing in under a minute on the average project. However, it is important to notice that in this study a mutation operator is blindly applied to any portion of a USC matching its mutation rules. Thus, any line of the USC is potentially subject to a fault. If practitioners have the possibility to foresee that their changes will affect only a specific portion of a USC, then they should leverage this information and properly reconsider the trade-offs on ***unique*** that we reported above.

For more comprehensive regression testing, Catana can expand the range of replayed transactions through policies like ***frequency*** or by increasing the budget for ***unique***. Methods that users invoke most frequently often include critical logic, and testing them more extensively (i.e., using a wider variety of transaction inputs and starting states) helps uncover more subtle contract breakages. The ***frequency*** policy, in particular, ensures that the core functions of a USC receive proportionate testing focus while still maintaining diversity across less-invoked methods as the test suite grows. However, practitioners should be mindful of diminishing returns when scaling up test suites. Beyond a certain threshold, additional tests may contribute only marginally while increasing execution time, regardless of the policy used. For instance, the ***unique*** policy raises the Mutation Score from 0.6 to approximately 0.72 within the first four expansions of the test suite, while it yields a more modest increase to 0.82 until the twentieth iteration.

Lastly, practitioners should be aware that selecting transactions exclusively based on recent user behavior (using the ***last*** policy) may limit test diversity. In scenarios where end-users interact with only a small subset of functions, ***last*** might produce a test suite that neglects certain methods, leaving portions of the contract untested. However, developers who anticipate newly emerging user behaviors can complement ***last*** with broader selection policies like ***unique*** or ***frequency***. Doing so ensures that new user interactions are not overlooked.

As a final note for practitioners, we remark that the policies discussed in this thesis only leverage structural information from smart contracts and their historical set of past transactions. This makes our guidelines broadly applicable when planning regression testing for any USC. At the same time, these policies do not take into account domain-specific behaviors or conditions that may influence testing priorities. Therefore, practitioners should consider how characteristics of their specific application might impact

the considerations we reported above.

11.6 Threats to Validity

In this Section, we discuss construct, internal, and external factors [147] that may affect the validity of this study.

Threats to Construct Validity Construct validity concerns the assumptions and decisions made during the design of the experiment that may influence the final results. One potential threat to construct validity stems from our method for extracting data on mapping variables. Catana identifies mapping elements by scraping Etherscan for storage slots that were historically impacted by replayed transactions. We made this choice because data scraping significantly reduces the time overhead of replay testing compared to analyzing the debug trace of a transaction. The drawback is that it narrows down the analysis to mapping slots that were accessed by the original transactions. As a result, if an upgrade introduces interactions with new mapping slots that were not previously accessed, Catana may not detect them. Future work should explore alternative approaches to avoid this kind of false negatives while keeping replay testing reasonably fast.

Another threat to validity is the risk of non-deterministic replays due to chain-related variables like the block timestamp. Although Catana inherits the forked blockchain's state, simulated transactions may still diverge from real blockchain execution. Catana mitigates this by discarding transactions that fail on the original contract, implicitly filtering out cases highly sensitive to execution context. Additionally, by replaying each transaction on both the original and upgraded contract within the same environment, Catana ensures any discrepancies affect both equally, isolating upgrade-induced differences from external factors.

Additionally, Catana uses Hardhat Network to setup the local blockchain environment in which to replay transactions. Similarly to other blockchain simulators, Hardhat Network allows forking the blockchain state only at the block level, rather than before the specific transaction itself. Therefore, if any two transactions t_1 and t_2 in the same block both target the contract under test, it might not be possible to correctly set-up the starting state for t_2. However, the probability of two related transactions being included in the same block is relatively low. Furthermore, Catana replays any transaction on the original contract to build the test oracle, which helps to mitigate this problem.

Threats to Internal Validity Internal validity is the extent to which the observed outcomes in our study can be confidently attributed to the variables under research. A threat to internal validity concerns the implementation of the experiment with respect to the policies *frequency* and *random*. Indeed, maintaining previously selected transactions in subsequent iterations of the experiment implies the selection of a small number of new transactions. In the case of *frequency*, this implementation strengthens the selection of the most frequently invoked methods, to the detriment of the others. The differences between the two policies may otherwise result less evident.

Threats to External Validity External validity indicates the extent to which the findings of the study can be generalized beyond the specific conditions of the experiment. The primary threat to external validity concerns the number of study subjects, which is limited by the high costs associated with the replay testing experiments. While using the full transaction history allows us to investigate the capabilities of replay testing in a comprehensive manner, the experiment is time-intensive and required us ≈ 75 days for all projects. The decision to cap transactions at 3,000 per project for computational feasibility may have influenced project selection, favoring newer contracts with lower activity. The insights provided in this thesis should be further explored on a variety of USCs, as specific characteristics of the considered projects and their usage patterns might influence the results we got.

11.7 Related Work

This thesis explored the use of capture-replay testing to evaluate the evolution of smart contracts. In this section, we review research closely related to our proposal, specifically focusing on smart contract testing, upgradeability, and capture-replay methodologies.

Smart Contract Testing Releasing reliable smart contract code is a primary concern for industry professionals working with blockchain [187]. So far, research in blockchain testing has primarily focused on stand-alone contracts, introducing techniques for defects and vulnerabilities detection, test generation and test adequacy assessment. To date, smart contract developers can rely on a range of static analysis tools (e.g., Oyente [103] and Slither [61]) and fuzzers (e.g., Echidna [70]) to disclose faults and optimize code prior to deployment. Solidity-specific mutation testing approaches were also proposed to improve the fault-detection capabilities of smart contract test suites (e.g., SuMo [22] and ReSuMo [24]). Several works do not specifically target the on-chain logic of a decentralized application (DApp), but consider the system as a whole. A few solutions build test cases for DApps [66, 195], or detect specific bugs stemming from the bad synchronization between on-chain and off-chain components [203].

Upgradeable Smart Contracts (USC) Despite their flexibility, Upgradeable Smart Contracts can introduce several risks. A few empirical studies summarize the characteristics of upgrade mechanisms and discuss their potential impact. Bodell et al. [79] developed a USC taxonomy comprising eleven unique upgradeability patterns and six related security issues. Li et al. [99] investigated the security implications of USC in the wild, covering 60M+ Ethereum smart contracts. Besides characterizing additional design patterns, they described several USC-specifc security issues, such as the lack of restrictive checks within upgradable functions. An empirical study by Liu et al. [101] shows that functionality update and addition are the two dominating intentions in upgrades, but introducing changes could impact impact client-specific compatibility. Indeed, they show that proxy-based contracts frequently encounter ABI-breaking changes, leading to real world broken usages. Qasse et al. [142] explored broader upgradeability concerns, showing that while upgrade patterns are mature and well-documented, their actual adoption

is more limited than expected. Although developers aim to implement feature improvements and security fixes, upgrading often leads to unintended consequences for contract reliability and user trust. Moreover, the impact on contract activity levels post-upgrade is a significant concern, as frequent or poorly communicated upgrades may lead to decreased usage.

Capture-Replay Testing for USCs The capture-replay approach has been widely utilized across various domains to ensure the continued correctness of software as it evolves [93, 9, 73, 80]. For example, in web testing [5, 102], it enables the validation of complex user interactions across different browsers and environments. In mobile testing [28, 143, 62, 69], it allows for the replication of touch gestures, swipes, and other user inputs across various devices. In IoT environments [58], it facilitates the verification of device behavior under diverse network conditions and real-world usage scenarios. In the blockchain domain, the inherent availability of transaction histories offers a unique opportunity for the validation of upgradeable smart contracts. However, no framework implements a systematic capture-replay testing approach for proxy-based systems. When considering compatibility issues, most available solutions only focus on the problem of detecting storage collisions. For example, the OpenZeppelin Upgrades plugin [128] implements static security checks to ensure upgrade safety, which does not consider behavioral compatibility. Even if storage remains intact, an upgrade can introduce semantic changes that alter the logic, break assumptions made by existing users or dependent contracts.

Instead, transaction replay has been considered in prior works, but none of these address the challenge of ensuring behavioral compatibility in upgradeable contracts. Available dynamic analysis frameworks that leverage historical transactions are designed for different purposes, including: *1) active monitoring*, *2) smart contract analysis*, *3) security analysis* and *4) test generation*. In the domain of active monitoring, DappGuard [46] intercepts real-time transactions to classify and learn potential attacks, and to protect smart contracts from them. Other tools use historical transactions for analysis and testing. ContractVis [74] generates test scripts from historical transactions and executes them in a minimal test environment. Unlike Catana, its purpose is to gain insights into the transparency of a smart contract with respect to its dependencies, and expose potential opacity issues. Instead, SmartGift [205] aims to generate tests for newly written smart contracts by extrapolating realistic test inputs from the transactions of other deployed contracts. Lastly, TxSpector [202] and EthScope [193] both provide transaction-level security analysis. TxSpector permits to investigate Ethereum transactions for attack detection. At a high level, it replays history transactions and records EVM bytecode-level traces to detect attacks that exploit various vulnerabilities (e.g.: Re-entrancy). EthScope defines a transaction-centric security analytics framework to detect malicious smart contracts on Ethereum. In this case, transaction replay is a means to re-execute suspicious transactions and confirm a malicious behavior.

11.8 Directions for Future Research

This Chapter introduced Catana, a novel replay-based regression testing framework for proxy-based Upgradeable Smart Contracts (USCs) on Ethereum. Catana captures trans-

actions from an already-deployed USC and harnesses them to validate a new logic contract version to be released. When a transaction is replayed on the smart contract under test, Catana checks both its output and its effects on storage, flagging any discrepancy with respect to the deployed USC. In this way, it allows practitioners to examine the behavior of an upgrade under real-world invocations before releasing it on Ethereum mainnet.

An important direction for future research involves expanding the study to cover a large dataset of USCs and evaluating Catana in real-world upgrade scenarios. While our evaluation simulated contract evolution using mutations, this approach inherently focuses on small, localized code changes. In practice, contract upgrades often involve more substantial modifications, such as the addition of entirely new features. To better assess the effectiveness of Catana, future research should investigate the detection of disruptive upgrades in real-world contract upgrade histories.

Lastly, although our analysis of selection policies showed that strategies emphasizing method-level diversity (e.g., the unique policy) perform well, there is room to refine these approaches. Future work could incorporate more advanced techniques, including the use of historical usage patterns, or the integration of Large Language Models (LLMs), to enhance the construction of replay test suites. Additionally, LLMs could be leveraged to automatically generate human-readable summaries of detected changes, identify potential root causes, and even propose targeted fixes for uncovered issues.

PART V

CONCLUSIONS

CHAPTER 12

CONCLUSIONS

12.1 Conclusions

The rapid growth of blockchain technology and the increasing adoption of smart contracts have brought significant opportunities but also substantial challenges in ensuring software quality. Smart contracts, being immutable and often handling valuable financial assets, demand a higher level of reliability and security than traditional software systems. This thesis addressed key challenges in smart contract quality across three critical phases: *testing*, *auditing*, and *maintenance*, with each research objective (RO) aligned to one of these phases.

RO1: Supporting Test Adequacy Assessment for Smart Contracts

With RO1, the thesis advanced the state of test adequacy assessment for smart contracts. It exposed the limits of traditional techniques and offered both practical insights and tools for mutation-based assessment, empowering developers to build more reliable Ethereum applications.

In particular, the thesis introduced SuMo, a mutation testing framework for Solidity smart contracts that can offer developers deeper insights into the effectiveness of their test suites. The tool is released as an open-source npm module and is fully compatible with all major Solidity testing frameworks, making it easy to integrate into existing projects. A key contribution lies in the design and implementation of 11 novel mutation operators, 7 of which are Solidity-specific and target previously unexplored language features (e.g., function overloading, function modifiers and global blockchain-related variables). The thesis empirically validated SuMo using open-source Solidity projects with high-coverage test suites. Empirical validation using high-coverage open-source projects showed that these Solidity-specific operators revealed significant blind spots in existing test suites, with a high number of surviving mutants. To improve usability and efficiency, SuMo also supports optimized operator variants, which reduce the number of subsumed mutants and cut testing time by around 30% without compromising the reliability of the adequacy assessment. Beyond adequacy assessment, SuMo demonstrated practical impact by guiding

test suite refinement, leading to a 53% increase in mutation score and the discovery of a real-world bug and faulty test cases.

To further integrate mutation testing within development workflows, the thesis introduced ReSuMo, the first *Regression Mutation Testing* (RMT) approach and tool for Solidity smart contracts. Built as an extension of SuMo, ReSuMo integrates regression testing and mutant selection techniques to enable rapid, incremental test adequacy evaluation in real-world development contexts. Its effectiveness was assessed across successive commits of active Ethereum projects, following their natural evolution on GitHub. Results show that ReSuMo maintained mutation testing accuracy while drastically reducing time costs. When mutant reuse was possible, ReSuMo achieved average time savings of 48% per project revision. In one complex case, testing time dropped from over 12 hours to just above 1 hour, making it possible to re-run mutation testing multiple times throughout the day. This approach makes it possible to incorporate mutation testing into tight feedback loops and continuous integration pipelines, enabling more agile and responsive test improvement practices.

RO2: Supporting Mutation Testing in Smart Contract Auditing

With RO2, the thesis demonstrated that mutation testing can play an important role in smart contract auditing by enhancing manual inspection processes, and even contributing to the discovery of issues through live mutant analysis and test generation.

To explore this potential, the thesis assessed the practical usefulness of live mutants (those undetected by existing test suites) for supporting code inspection during audits. A study was conducted on real-world Solidity projects audited by Quantstamp, examining the relationship between live mutants and issues flagged by professional auditors. Results revealed that approximately 67% of the analyzed issues were associated with productive mutants, and over a third (34.2%) of those mutants were deemed potentially valuable for code inspection. These findings suggest that live mutants are not only effective in guiding test suite improvements but can also support auditors in identifying subtle, code-level issues that might otherwise be overlooked. This expands the utility of mutation testing beyond test assessment, positioning it as a tool for augmenting human reasoning during audits.

To operationalize these insights, the thesis introduced an end-to-end mutation testing framework designed to fit auditing workflows. This framework enables the remote execution of mutation testing tasks and jobs, allowing auditors to delegate test execution to the cloud and obtain the results earlier, while they can still be useful for the audit. Five different strategies for distributing mutation jobs were evaluated, and practical guidelines were developed to help auditors select optimal configurations. Performance measurements showed that the most effective strategy can cut down mutation testing time by up to 70%, transforming a traditionally resource-intensive task into a manageable component of the audit cycle.

Finally, the thesis presented Alchemist, a novel method that leverages Large Language Models (LLMs) to generate test cases from undetected mutants. Unlike direct LLM prompting, Alchemist integrates the scientific method into the test generation loop, systematically formulating and refining test hypotheses based on mutant behavior. Evaluations on production-grade Solidity projects showed that this guided process outperforms

naive prompting in both code correctness and mutant-killing effectiveness, yielding higher Mutation Scores. Alchemist can automate the generation of meaningful test cases for smart contracts, reducng the manual burden typically associated with mutation analysis, and making the approach more accessible and impactful in auditing scenarios.

RO3: Supporting Smart Contract Maintenance Practices

With RO3, the thesis addressed the lack of advanced solutions for validating upgrades of proxy-based Ethereum smart contracts (USCs). In particular, the thesis introduced Catana, the first regression testing framework for USCs based on historical transaction replay. Unlike existing solutions that primarily focus on static storage layout checks, Catana captures transactions from an already-deployed USC and harnesses them to validate a new logic contract version to be released, supporting practitioners in three main areas: (1) *detecting disruptive changes* that might indicate regressions or compatibility issues, (2) *validating intentional modifications*, ensuring they align with expectations, and (3) *debugging*, providing detailed pre- and post-upgrade test logs. The framework is also proxy-agnostic, making it compatible with widely adopted standards such as Transparent and UUPS proxies. Additionally, Catana improves testing observability by comparing both transaction outputs and storage modifications across contract versions. Experimental results showed that this fine-grained analysis is essential, as 86.5% of the faults introduced in upgraded versions were only detectable through storage state comparisons. Furthermore, the thesis evaluated four transaction selection strategies, demonstrating that those prioritizing method-level diversity (e.g., `unique`) consistently achieved higher mutation scores while minimizing the number of required replays. Notably, even compact test suites (e.g., containing a single transaction per method) were able to detect around 60% of disruptive changes, showing the feasibility of lightweight but effective replay-based testing. Overall, this contribution supports more reliable maintenance of Ethereum applications as their underlying logic evolves.

BIBLIOGRAPHY

[1] Sefa Akca, Chao Peng, and Ajitha Rajan. Testing smart contracts: Which technique performs best? In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2021.

[2] Nadir Akhtar. What is a Smart Contract Audit? `https://quantstamp.com/blog/what-is-a-smart-contract-audit`.

[3] Erfan Andesta, Fathiyeh Faghih, and Mahdi Fooladgar. Testing smart contracts gets smarter. In *2020 10th International Conference on Computer and Knowledge Engineering ICCKE*, pages 405–412. IEEE, 2020.

[4] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[5] Silviu Andrica and George Candea. Warr: A tool for high-fidelity web application record and replay. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 403–410. IEEE Compute Society, 2011.

[6] Monika Di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *IEEE International Conference on Decentralized Applications and Infrastructures, DAPPCON 2019, Newark, CA, USA, April 4-9, 2019*, pages 69–78. IEEE, 2019.

[7] Hilary Arksey and Lisa O'Malley. Scoping studies: towards a methodological framework. *International Journal of Social Research Methodology*, 8(1):19–32, 2005.

[8] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.

[9] Majid Babaei and Juergen Dingel. Efficient replay-based regression testing for distributed reactive systems in the context of model-driven development. In *2021*

ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), pages 89–100. IEEE, October 2021.

[10] Sebastian Banescu, Morena Barboni, Andrea Morichetta, Andrea Polini, and Edward Zulkoski. Enhanced mutation testing of smart contracts in support of code inspection. In *IEEE International Conference on Blockchain and Cryptocurrency ICBC*, pages 558–566. IEEE, 2024.

[11] Fadi Barbàra, Emanuele A Napoli, Valentina Gatteschi, and Claudio Schifanella. Automatic smart contract generation through llms: When the stochastic parrot fails. In *6th Distributed Ledger Technology Workshop*, 2024.

[12] Morena Barboni, Guglielmo De Angelis, Andrea Morichetta, and Andrea Polini. CATANA: replay testing for the ethereum blockchain. In *Testing Software and Systems - 35th IFIP WG 6.1 International Conference ICTSS 2023*, volume 14131 of *Lecture Notes in Computer Science*, pages 257–265. Springer, 2023.

[13] Morena Barboni, Antonia Bertolino, and Guglielmo De Angelis. What we talk about when we talk about software test flakiness. In *14th International Conference on the Quality of Information and Communications Technology QUATIC 2021*, volume 1439, pages 29–39. Springer, 2021.

[14] Morena Barboni, Antonia Bertolino, and Guglielmo De Angelis. Flakiness goes live: Insights from an in vivo testing simulation study. *Information and Software Technology*, 167:107373, 2024.

[15] Morena Barboni, Francesco Casoni, Andrea Morichetta, and Andrea Polini. Resumo: Regression mutation testing for solidity smart contracts. In *15th International Conference on the Quality of Information and Communications Technology QUATIC 2022*, volume 1621, pages 61–76. Springer, 2022.

[16] Morena Barboni, Guglielmo De Angelis, Andrea Morichetta, and Andrea Polini. Wielding blockchain transactions for capture-replay testing of upgradeable smart contracts. *ACM Transactions on Internet Technology*, 2025.

[17] Morena Barboni, Filippo Lampa, Andrea Morichetta, Andrea Polini, and Edward Zulkoski. Alchemist: Llm-driven test generation using solidity mutants and the scientific method. In *IEEE International Conference on Blockchain and Cryptocurrency ICBC 2025*, 2025.

[18] Morena Barboni, Filippo Lampa, Andrea Morichetta, Andrea Polini, and Edward Zulkoski. Mutant-driven test generation for ethereum smart contracts via llms. In *IEEE International Conference on Artificial Intelligence Testing, AITest*, 2025.

[19] Morena Barboni, Andrea Morichetta, and Andrea Polini. Sumo: A mutation testing strategy for solidity smart contracts. In *2nd IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2021*, pages 50–59. IEEE, 2021.

[20] Morena Barboni, Andrea Morichetta, and Andrea Polini. Smart contract testing: challenges and opportunities. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2022*, pages 21–24. ACM, 2022.

[21] Morena Barboni, Andrea Morichetta, and Andrea Polini. Smart contract testing: challenges and opportunities. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 21–24. ACM, 2022.

[22] Morena Barboni, Andrea Morichetta, and Andrea Polini. Sumo: A mutation testing approach and tool for the ethereum blockchain. *Journal of Systems and Software*, 193:111445, 2022.

[23] Morena Barboni, Andrea Morichetta, Andrea Polini, Sebastian Banescu, and Edward Zulkoski. Mutation testing of smart contracts as a service. In *17th International Conference on the Quality of Information and Communications Technology QUATIC 2024*, volume 2178, pages 93–109. Springer, 2024.

[24] Morena Barboni, Andrea Morichetta, Andrea Polini, and Francesco Casoni. Resumo: a regression strategy and tool for mutation testing of solidity smart contracts. *Software Quality Journal*, 32(1):225–253, 2024.

[25] Gabriele Bavota and Barbara Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE International Conference on Software Maintenance and Evolution ICSME 2015*, pages 81–90. IEEE Computer Society, 2015.

[26] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry - A study at facebook. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021*, pages 268–277. IEEE, 2021.

[27] Chaïmaa Benabbou and Önder Gürcan. A survey of verification, validation and testing solutions for smart contracts. In *2021 Third International Conference on Blockchain Computing and Applications (BCCA)*, pages 57–64. IEEE, 2021.

[28] Carlo Bernaschina, Roman Fedorov, Darian Frajberg, and Piero Fraternali. A framework for regression testing of outdoor mobile applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 179–181. IEEE, 2017.

[29] Antonia Bertolino, Guglielmo De Angelis, and Francesca Lonetti. Governing regression testing in systems of systems. In *IEEE ISSRE Workshops*, pages 144–148. IEEE, 2019.

[30] Antonia Bertolino, Guglielmo De Angelis, Breno Miranda, and Paolo Tonella. Unobtrusive in vivo test and rollback of Java applications. Technical report, ISTI-2022-TR/008, 2022, 2022.

[31] Antonia Bertolino, Guglielmo De Angelis, Breno Miranda, and Paolo Tonella. In vivo test and rollback of Java applications as they are. *Software Testing, Verification and Reliability*, 33(7):e1857, 2023.

[32] Konstantin Britikov, Ilia Zlatkin, Grigory Fedyukovich, Leonardo Alt, and Natasha Sharygina. Soltg: A chc-based solidity test case generator. In *Computer Aided Verification - 36th International Conference, CAV 2024*, volume 14681 of *Lecture Notes in Computer Science*, pages 466–479. Springer, 2024.

[33] Timothy A Budd, Richard J Lipton, Richard A DeMillo, and Frederick G Sayward. *Mutation analysis*. Yale University. Department of Computer Science, 1979.

[34] Van Cuong Bui, Sheng Wen, Jiangshan Yu, Xin Xia, Mohammad Sayad Haghighi, and Yang Xiang. Evaluating upgradable smart contract. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 252–256. IEEE, 2021.

[35] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[36] Mark Anthony Cachia, Mark Micallef, and Christian Colombo. Towards incremental mutation testing. *Electronic Notes in Theoretical Computer Science*, 294:2–11, 2013.

[37] Pablo C. Cañizares, Alberto Núñez, Rosa Filgueira, and Juan de Lara. Parallel mutation testing for large scale systems. *Clust. Comput.*, 27(2):2071–2097, 2024.

[38] Davide G. Cavezza, Roberto Pietrantuono, Javier Alonso, Stefano Russo, and Kishor S. Trivedi. Reproducibility of environment-dependent software failures: An experience report. In *Proc. IEEE ISSRE 2014, Naples, Italy*, pages 267–276. IEEE, 2014.

[39] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. Smart contract and defi security tools: Do they meet the needs of practitioners? In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*, pages 60:1–60:13. ACM, 2024.

[40] P. Chapman, D. Xu, L. Deng, and Y. Xiong. Deviant: A mutation testing tool for solidity smart contracts. In *International Conference on Blockchain*, pages 319–324. IEEE, 2019.

[41] Siddhartha Chatterjee and Bina Ramamurthy. Efficacy of various large language models in generating smart contracts. *arXiv preprint arXiv:2407.11019*, 2024.

[42] Lingchao Chen and Lingming Zhang. Speeding up mutation testing via regression test selection: An extensive study. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 58–69. IEEE, 2018.

[43] Code4rena. Code4rena. `https://code4rena.com/`.

[44] Consensys. Surya: A set of utilities for exploring Solidity contracts. https://github.com/Consensys/surya.

[45] Consensys. Truffle Suite. https://archive.trufflesuite.com/, 2022.

[46] Thomas Cook, Alex Latham, and Jae Hyung Lee. Dappguard : Active monitoring and defense for solidity smart contracts. Online on https://courses.csail.mit.edu/6.857/2017/project/23.pdf, 2017.

[47] Domenico Cotroneo, Michael Grottke, Roberto Natella, Roberto Pietrantuono, and Kishor S. Trivedi. Fault triggers in open-source software: An experience report. In *Proc. IEEE ISSRE 2013, Pasadena, CA, USA*, pages 178–187. IEEE, 2013.

[48] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468, 2024.

[49] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338*, 2023.

[50] Lin Deng, Jeff Offutt, and Nan Li. Empirical evaluation of the statement deletion mutation operator. In *International Conference on Software Testing, Verification and Validation*, pages 84–93. IEEE, 2013.

[51] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert Hierons. Smart contracts vulnerabilities: a call for blockchain software engineering? In *Int. Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 19–25. IEEE, 2018.

[52] Kadiatou Diallo, Zizhao Chen, W. Eric Wong, and Shou-Yu Lee. An analysis and comparison of mutation testing tools for python. In *11th International Conference on Dependable Systems and Their Applications DSA 2024*, pages 161–169. IEEE, 2024.

[53] Ardit Dika and Mariusz Nowostawski. Security vulnerabilities in ethereum smart contracts. In *Internet of Things and Green Computing and Communications and Cyber, Physical and Social Computing and Smart Data*, pages 955–962. IEEE, 2018.

[54] Stefan Driessen, Dario Di Nucci, Geert Monsieur, Damian A. Tamburri, and Willem-Jan van den Heuvel. Automated test-case generation for solidity smart contracts: the agsolt approach and its evaluation, 2022.

[55] Stefan W. Driessen, Dario Di Nucci, Damian A. Tamburri, and Willem-Jan van den Heuvel. Solar: Automated test-suite generation for solidity smart contracts. *Sci. Comput. Program.*, 232:103036, 2024.

[56] erigontech. Erigon: Ethereum implementation on the efficiency frontier. https://github.com/erigontech/erigon.

[57] eth brownie. Brownie: A Python-based development and testing framework for smart contracts targeting the Ethereum Virtual Machine. `https://github.com/eth-brownie/brownie`.

[58] Kaiming Fang and Guanhua Yan. Iotreplay: Troubleshooting cots iot devices with record and replay. In *2020 IEEE/ACM Symposium on Edge Computing*, pages 193–205. IEEE, 2020.

[59] David Faragó. Engineering A reliable prompt for generating unit tests - prompt engineering for QA & QA for prompt engineering. *Softwaretechnik-Trends*, 43(3):10–14, 2023.

[60] FedericoBond. eth-mutants: A mutation testing tool for Solidity contracts. `https://github.com/federicobond/eth-mutants`.

[61] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 8–15. IEEE/ACM, 2019.

[62] Sidong Feng and Chunyang Chen. Gifdroid: automated replay of visual bug reports for android apps. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1045–1057. ACM, 2022.

[63] foundry rs. Foundry: Foundry is a blazing fast, portable and modular toolkit for Ethereum application development written in Rust. `https://github.com/foundry-rs/foundry`.

[64] Phyllis G. Frankl, Stewart N. Weiss, and Cang Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[65] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 147–158, 2010.

[66] Jianbo Gao. Guided, automated testing of blockchain-based decentralized applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings*, pages 138–140. IEEE/ACM, 2019.

[67] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Ekstazi: Lightweight test selection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 713–716. IEEE, 2015.

[68] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015*, pages 211–222. ACM, 2015.

[69] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing-and touch-sensitive record and replay for android. In *35th International Conference on Software Engineering, ICSE '13*, pages 72–81. IEEE Computer Society, 2013.

[70] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560. ACM, 2020.

[71] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In *Financial Cryptography and Data Security*, pages 634–653. Springer International Publishing, 2020.

[72] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018.

[73] Mouna Hammoudi. Regression testing of web applications using record/replay tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'16, pages 1079–1081. ACM, November 2016.

[74] Pieter Hartel and Mark van Staalduinen. Truffle tests for free–replaying ethereum smart contracts for transparency. arXiv preprint arXiv:1907.09208, 2019.

[75] Pieter H. Hartel and Richard Schumi. Mutation testing of smart contracts at scale. In *International Conference TAP@STAF 2020*, volume 12165 of *LNCS*, pages 23–42. Springer, 2020.

[76] Joran J. Honig, Maarten H. Everts, and Marieke Huisman. Practical mutation testing for smart contracts. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 289–303. Springer, 2019.

[77] Shamaila Hussain. Mutation clustering. *Ms. Th., Kings College London, Strand, London*, 9, 2008.

[78] IEEE. IEEE standard for software reviews and audits. *IEEE Std 1028-2008*, pages 1–53, 2008.

[79] William Edward Bodell III, Sajad Meisami, and Yue Duan. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium, 2023*, pages 1829–1846. USENIX Association, 2023.

[80] Javaria Imtiaz, Muhammad Zohaib Iqbal, and Muhammad Uzair Khan. An automated model-based approach to repair test suites of evolving web applications. *Journal of Systems and Software*, 171:110841, 2021.

[81] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *36th International Conference on Software Engineering, ICSE 2014*, pages 435–445. ACM, 2014.

[82] ISO/IEC 25010. `https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-2:v1:en`, 2023.

[83] ISTQB Glossary. `https://glossary.istqb.org/en_US/home`.

[84] Y Ivanova and A Khritankov. Regularmutator: a mutation testing tool for solidity smart contracts. *Procedia Computer Science*, 178:75–83, 2020.

[85] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009. Source Code Analysis and Manipulation SCAM 2008.

[86] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[87] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[88] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[89] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *Empirical Software Engineering*, 30(2):1–28, 2025.

[90] Niclas Kannengießer, Sebastian Lins, Christian Sander, Klaus Winter, Hellmuth Frey, and Ali Sunyaev. Challenges and common solutions in smart contract development. *IEEE Transactions on Software Engineering*, 48(11):4291–4318, 2022.

[91] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *17th Asia Pacific Software Engineering Conference APSEC 2010*, pages 300–309. IEEE Computer Society, 2010.

[92] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. Modeling and ranking flaky tests at Apple. In *Proc. ICSE-SEIP*, pages 110–119. ACM, 2020.

[93] Ilia Kravets and Dan Tsafrir. Feasibility of mutable replay for automated regression testing of security updates. In *Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*. IEEE, 2012.

[94] Bob Kurtz, Paul Ammann, Márcio Eduardo Delamaro, Jeff Offutt, and Lin Deng. Mutant subsumption graphs. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings*, pages 176–185. IEEE Computer Society, 2014.

[95] Wing Lam. IDoFT Home Page. https://mir.cs.illinois.edu/flakytests.

[96] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering FSE 2016*, pages 583–594. ACM, 2016.

[97] Owolabi Legunsen, August Shi, and Darko Marinov. Starts: Static regression test selection. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 949–954. IEEE, 2017.

[98] Nan Li, Michael West, Anthony Escalona, and Vinicius H. S. Durelli. Mutation testing in practice using ruby. In *Proc. ICST*, pages 1–6. IEEE Computer Society, 2015.

[99] Xiaofan Li, Jin Yang, Jiaqi Chen, Yuzhe Tang, and Xing Gao. Characterizing ethereum upgradable smart contracts and their security implications. In *Proceedings of the ACM on Web Conference 2024*, pages 1847–1858. ACM, 2024.

[100] Zixin Li, Haoran Wu, Jiehui Xu, Xingya Wang, Lingming Zhang, and Zhenyu Chen. MuSC: A tool for mutation testing of ethereum smart contract. *International Conference on Automated Software Engineering ASE 2019*, pages 1198–1201, 2019.

[101] Ye Liu, Shuo Li, Xiuheng Wu, Yi Li, Zhiyang Chen, and David Lo. Demystifying the characteristics for smart contract upgrades. arXiv preprint arXiv:2406.05712, 2024.

[102] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. Webrr: self-replay enhanced robust record/replay for web application testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1498–1508. ACM, 2020.

[103] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269. ACM, 2016.

[104] Wei Ma, Thomas Laurent, Miloš Ojdanić, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. Commit-aware mutation testing. In *2020 IEEE International Conference on Software Maintenance and Evolution ICSME*, pages 394–405. IEEE, 2020.

[105] Yu-Seung Ma and Jeff Offutt. Description of class mutation mutation operators for java. https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf.

[106] Yu-Seung Ma and Jeff Offutt. Description of mujava's method-level mutation operators. https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf.

[107] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[108] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In *Proc. ICSE-SEIP*, pages 91–100. IEEE, 2019.

[109] Lodovica Marchesi, Michele Marchesi, Livio Pompianu, and Roberto Tonelli. Security checklists for ethereum smart contract development: patterns and best practices. *arXiv preprint arXiv:2008.04761*, 2020.

[110] Pedro Reales Mateo and Macario Polo Usaola. Parallel mutation testing. *Software Testing, Verification and Reliability*, 23(4):315–350, 2013.

[111] Aditya P Mathur. *Foundations of software testing*. Pearson Education, Delhi, India, 2 edition, 2014.

[112] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*, pages 523–540. Springer, 2018.

[113] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain. *Journal of Cases on Information Technology*, 21(1):19–32, 2019.

[114] Robert G. Merkel and James Georgeson. Cloud-based distributed mutation analysis. *CoRR*, abs/1601.07157, 2016.

[115] John Micco. Flaky tests at Google and how we mitigate them, 2016.

[116] Mahdi H. Miraz and Maaruf Ali. Blockchain enabled smart contract based applications: Deficiencies with the software development life cycle models. *Information Systems eJournal*, 2020.

[117] MorenaBarboni. Alchemist: Alchemist Test Generator. `https://github.com/MorenaBarboni/Alchemist`.

[118] MorenaBarboni. Catana: Capture-Replay Testing for USCs. `https://github.com/MorenaBarboni/Catana`.

[119] MorenaBarboni. Catana_ReplicationPackage: Replication Package for the mutation testing experiments with Catana. `https://github.com/MorenaBarboni/Catana_ReplicationPackage`.

[120] MorenaBarboni. @morenabarboni/sumo. `https://www.npmjs.com/package/%40morenabarboni/sumo`.

[121] MorenaBarboni. SuMo-SOlidity-MUtator: A mutation testing tool for Solidity Smart Contracts. `https://github.com/MorenaBarboni/SuMo-SOlidity-MUtator`.

[122] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[123] Dominik Muhs. Smart Contract Weakness Classification (SWC). `https://swcregistry.io/`.

[124] Nomic Foundation. Hardhat | Ethereum development environment for professionals by Nomic Foundation. `https://hardhat.org/`, 2025.

[125] A. Jefferson Offutt, Roy P. Pargas, Scott V. Fichter, and Prashant K. Khambekar. Mutation testing of software using MIMD computer. In *Proc. ICPP*, pages 257–266. CRC Press, 1992.

[126] Mitchell Olsthoorn, Dimitri Michel Stallenberg, Arie van Deursen, and Annibale Panichella. Syntest-solidity: Automated test case generation and fuzzing for smart contracts. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion*, pages 202–206. ACM/IEEE, 2022.

[127] OpenZeppelin. Proxies - OpenZeppelin Docs. `https://docs.openzeppelin.com/contracts/5.x/api/proxy`, 2025.

[128] OpenZeppelin. Upgrades Plugins - OpenZeppelin Docs. `https://docs.openzeppelin.com/upgrades-plugins/`, 2025.

[129] Wendkûuni C. Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. Large-scale, independent and comprehensive study of the power of llms for test case generation, 2024.

[130] Luisa Palechor and Cor-Paul Bezemer. How are solidity smart contracts tested in open source projects? an exploratory study. In *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022*, pages 165–169. ACM/IEEE, 2022.

[131] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. Multi-language unit test generation using llms, 2024.

[132] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering SANER 2015*, pages 161–170. IEEE Computer Society, 2015.

[133] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, pages 936–946. IEEE Computer Society, 2015.

[134] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. *Advances in Computers*, 112:275–378, 2019.

[135] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice ICSE (SEIP) 2018*, pages 163–171. ACM, 2018.

[136] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Does mutation testing improve testing practices? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 910–921. IEEE, 2021.

[137] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering*, 48(10):3900–3912, 2022.

[138] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. Please fix this mutant: How do developers resolve mutants surfaced during code review? In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice ICSE (SEIP) 2023*, pages 150–161. IEEE, 2023.

[139] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops ICSTW*, pages 47–53. IEEE, 2018.

[140] PIT Mutation Testing. `https://pitest.org`.

[141] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leonardo Fernandes, and Márcio Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157, 2019.

[142] Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson. Immutable in principle, upgradeable by design: Exploratory study of smart contract upgradeability. arXiv preprint arXiv:2407.01493, 2024.

[143] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, pages 571–582. ACM, 2016.

[144] Quantstamp. Quantstamp: Securing the Future of Web3. `https://quantstamp.com/`.

[145] Quantstamp. Quantstamp 2023 Web3 Security Year In Review. `https://quantstamp.com/blog/quantstamp-2023-web3-security-year-in-review`, 2024.

[146] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. Not your type! detecting storage collision vulnerabilities in ethereum smart contracts. In *31st Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2024.

[147] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14:131–164, 2009.

[148] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971, 2024.

[149] Mehdi Salehi, Jeremy Clark, and Mohammad Mannan. Not so immutable: Upgradeability of smart contracts on ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 539–554. Springer, Springer, 2022.

[150] sc forks. solidity-coverage: Code coverage for Solidity smart-contracts. `https://github.com/sc-forks/solidity-coverage`.

[151] August Shi, Jonathan Bell, and Darko Marinov. Mitigating the effects of flaky tests on mutation testing. In *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 296–306. ACM, 2019.

[152] August Shi, Jonathan Bell, and Darko Marinov. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 112–122, 2019.

[153] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 313–322, 2024.

[154] IEEE Computer Society. What is Software Quality? `https://www.computer.org/resources/what-is-software-quality`.

[155] Block and transaction properties - Solidity 0.7.1 documentation. `https://docs.soliditylang.org/en/v0.7.1/units-and-global-variables.html#block-and-transaction-properties`.

[156] Constructor - Solidity 0.7.1 documentation. `https://docs.soliditylang.org/en/v0.7.1/contracts.html#constructors`.

[157] Contract related - Solidity 0.7.1 documentation. `https://docs.soliditylang.org/en/v0.7.1/units-and-global-variables.html#contract-related`.

[158] Data location - Solidity 0.7.1 documentation. `https://docs.soliditylang.org/en/v0.7.1/types.html#data-location`.

[159] Error handling - Solidity 0.7.1 documentation. `https://docs.soliditylang.org/en/v0.7.1/control-structures.html#assert-and-require`.

[160] Events - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#events.

[161] Expressions and control structures - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/control-structures.html.

[162] Function oveloading - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#function-overloading.

[163] Function overriding - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#function-overriding.

[164] Funtion parameters and return variables - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#function-parameters-and-return-variables.

[165] Inheritance - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#inheritance.

[166] Libraries - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#libraries.

[167] Mathematical and cryptographic functions - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/units-and-global-variables.html#mathematical-and-cryptographic-functions.

[168] Members of address types - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/units-and-global-variables.html#members-of-address-types.

[169] Modifier overriding - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#modifier-overriding.

[170] Modifiers - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#function-modifiers.

[171] Operators cheatsheet - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/cheatsheet.html#order.

[172] Return variables - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#return-variables.

[173] Returning multiple values - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html#returning-multiple-values.

[174] Solidity Language grammar - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/grammar.html.

[175] Type conversions - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/types.html#conversions-between-elementary-types.

[176] Types - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/types.html.

[177] Units and globally available variables - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/units-and-global-variables.html?highlight=gwei.

[178] Visibility and getters - Solidity 0.7.1 documentation. https://docs.soliditylang.org/en/v0.7.1/contracts.html?highlight=modifier#visibility-and-getters.

[179] Ju Myung Song, Jongwook Sung, and Taeho Park. Applications of blockchain to improve supply chain traceability. *Procedia Computer Science*, 162:119–122, 2019.

[180] Chainalysis Team. 2022 Biggest Year Ever For Crypto Hacking. https://www.chainalysis.com/blog/2022-biggest-year-ever-for-crypto-hacking/, 2023.

[181] Cyfrin Team. What is a Smart Contract Security Audit? - Full Guide. https://www.cyfrin.io/blog/what-is-a-smart-contract-security-audit, 2024.

[182] David Tengeri, Laszlo Vidacs, Arpad Beszedes, Judit Jasz, Gergo Balogh, Bela Vancsics, and Tibor Gyimothy. Relating code coverage, mutation score and test suite reducibility to defect density. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops ICSTW*. IEEE, 2016.

[183] TIOBE. TIOBE Index. https://www.tiobe.com/tiobe-index/.

[184] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 67–82, 2018.

[185] Anna Vacca, Andrea Di Sorbo, Corrado A Visaggio, and Gerardo Canfora. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *Journal of Systems and Software*, 174:110891, 2021.

[186] Fernando Richter Vidal, Naghmeh Ivaki, and Nuno Laranjeiro. Analyzing the impact of elusive faults on blockchain reliability. *CoRR*, abs/2304.05520, 2023.

[187] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. Smart contract security: A practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*, pages 1410–1422. IEEE, 2021.

[188] Xinming Wang, Zhijian Xie, Jiahao He, Gansen Zhao, and Ruihua Nie. Basis path coverage criteria for smart contract application testing. In *2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 34–41. IEEE, 2019.

[189] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[190] Elaine J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, 31(6):668–675, 1988.

[191] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger byzantium version, 2019.

[192] Haoran Wu, Xingya Wang, Jiehui Xu, Weiqin Zou, Lingming Zhang, and Zhenyu Chen. Mutation testing for ethereum smart contract. *ArXiv*, abs/1908.03707, 2019.

[193] Lei Wu, Siwei Wu, Yajin Zhou, Runhuai Li, Zhi Wang, Xiapu Luo, Cong Wang, and Kui Ren. Ethscope: A transaction-centric security analytics framework to detect malicious smart contracts on ethereum. arXiv preprint arXiv:2005.08278, 2020.

[194] Siwei Wu, Lei Wu, Yajin Zhou, Runhuai Li, Zhi Wang, Xiapu Luo, Cong Wang, and Kui Ren. Time-travel investigation: Toward building a scalable attack detection framework on ethereum. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–33, 2022.

[195] Zhenhao Wu, Jiashuo Zhang, Jianbo Gao, Yue Li, Qingshan Li, Zhi Guan, and Zhong Chen. Kaya: A testing framework for blockchain-based decentralized applications. In *2020 IEEE International Conference on Software Maintenance and Evolution*, pages 826–829. IEEE, 2020.

[196] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

[197] Andreas Zeller. *Why programs fail: a guide to systematic debugging.* Morgan Kaufmann, 2009.

[198] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Faulttracer: a change impact and regression fault analysis tool for evolving java programs. In *20th ACM Symposium on the Foundations of Software Engineering SIGSOFT/FSE 2012*, page 40. ACM, 2012.

[199] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*. ACM Press, 2012.

[200] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 331–341, 2012.

[201] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering ICSE 2010*, pages 435–444. ACM, 2010.

[202] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. TXSPECTOR: uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium*, pages 2775–2792. USENIX Association, 2020.

[203] Wuqi Zhang, Lili Wei, Shuqing Li, Yepang Liu, and Shing-Chi Cheung. Ðarcher: Detecting on-chain-off-chain synchronization bugs in decentralized applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 553–565. ACM, 2021.

[204] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering ICSE*, page 615–627. IEEE, 2023.

[205] Teng Zhou, Kui Liu, Li Li, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. Smartgift: Learning to generate practical inputs for testing smart contracts. In *2021 IEEE International Conference on Software Maintenance and Evolution*, pages 23–34. IEEE, 2021.

[206] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[207] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D. Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 47:2084–2106, 2021.

[208] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 47(10):2084–2106, 2019.