UNICAM
Università di Camerino
1336

Università di Camerino

SuMo

Mutation Testing for Solidity Contracts

Writing Better Tests with SuMo

1. **SUT: Running Example**:
   - Introduction to the SUT;
2. **Code Coverage: Solidity-Coverage**
   - What it tells us about our tests;
   - Why it doesn't tell the whole story.
3. **Mutation Testing: SuMo:**
   - Introduction to SuMo;
   - Running Mutation Testing;
   - Analyzing Live Mutants.

Demo Repository

morenabarboni/sumo-demo

SuMo-Demo

In the repository you will find:

- `CampusCoin Setup`
- `Test Code Examples`
- `Slides`

Prerequisites

Ensure the following are installed:

- `Node.js`
- `npm` (comes with Node.js)

SUT: Business Logic

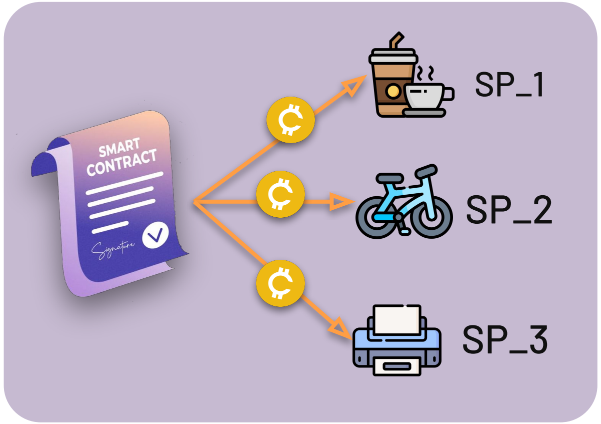
contracts/CampusCoin.sol: A custom **ERC-20 token** that can be used to pay for services around campus (e.g., buying food, borrowing books).

Active Roles:

- **Admin:** Registers Users and Mints new CC;
- **Student**: Pays Service Providers using CC;

Passive Roles:

- **ServiceProvider:** Receives CC payments;
- **University:** Receives a 1% fee on payments.



SUT: Test Environment

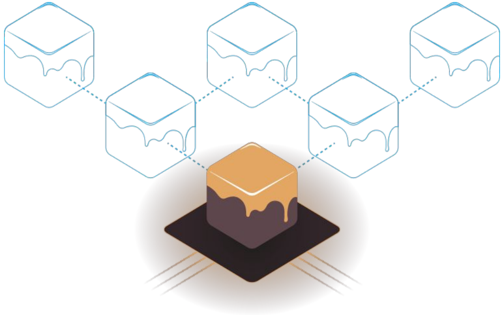
test/CampusCoin.js: The HardHat test file for CampusCoin.sol

1) Testing Framework

Provides tools and domain-specific utilities to write, organize and execute test cases.

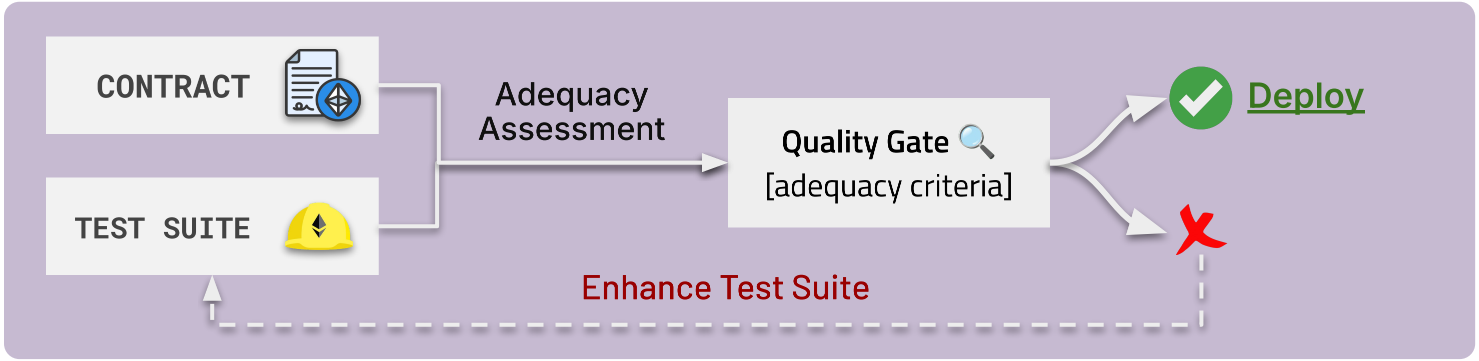
2) Chain Simulator

Creates a **local blockchain environment** to deploy and test contracts without costs.



Goal

Does the CampusCoin.js test suite gives us **sufficient confidence** in the **correctness** of the CampusCoin.sol Smart Contract?



- If the TS meets **adequacy criteria**, the Contract passes the QG;
- Different adequacy criteria = different deployment decisions.

Quality Gate Based on Coverage Criteria

To ensure **high confidence** in the **correctness** of our contract, we define **strict coverage thresholds** as part of our deployment quality gate:

Statement Coverage == 100%

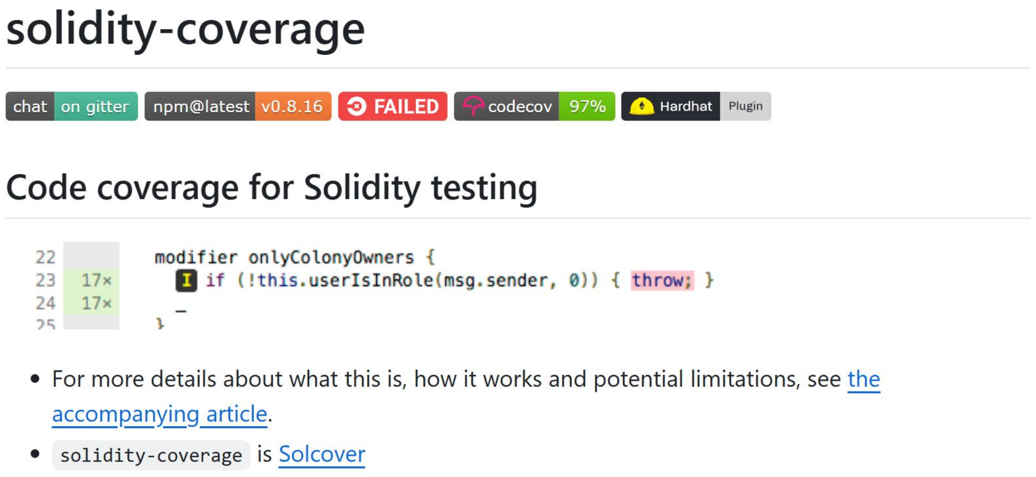
Every statement in the contract must be executed at least once.

Branch Coverage == 100%

Every branch (e.g., require) must be evaluated in both directions.

For most contracts, this is feasible and cheaper than the cost of failure.

Coverage Analysis with HardHat



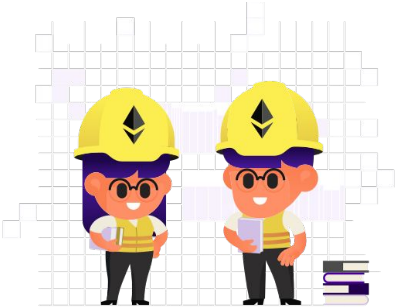
solidity-coverage

chat on gitter | npm@latest v0.8.16 | FAILED | codecov 97% | Hardhat Plugin

Code coverage for Solidity testing

```
        modifier onlyColonyOwners {
17×       I if (!this.userIsInRole(msg.sender, 0)) { throw; }
17×       _
        }
```

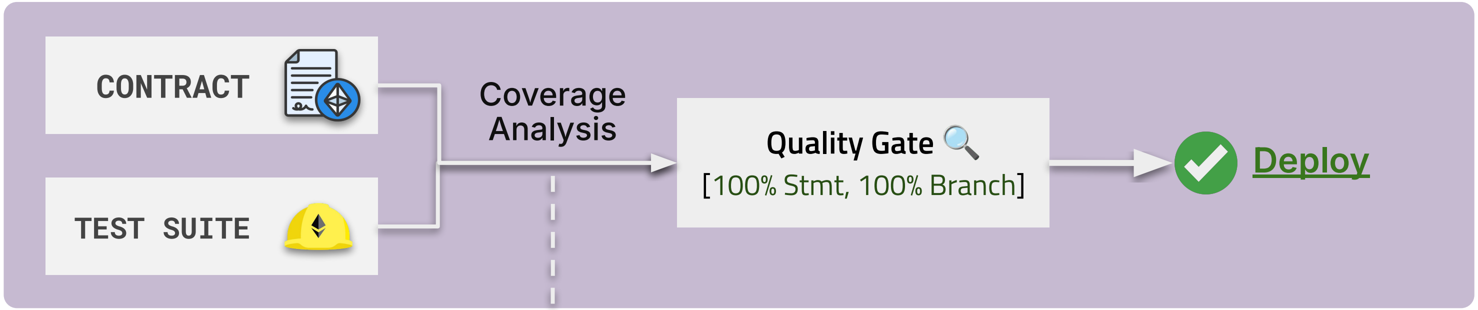
- For more details about what this is, how it works and potential limitations, see the accompanying article.
- `solidity-coverage` is Solcover



👉 **Let's try it out: npx hardhat coverage**

Coverage Analysis: Results

Our test suite meets the adequacy criteria, looks like we're done…



CONTRACT

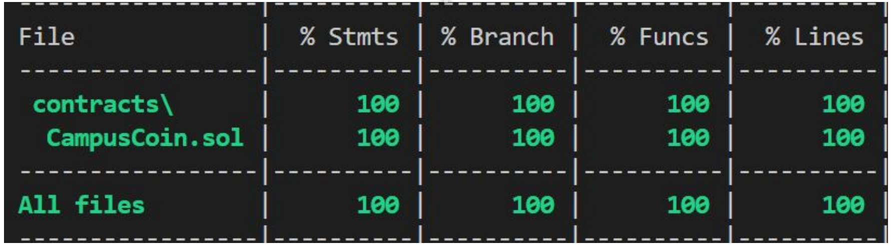
TEST SUITE

Coverage Analysis

Quality Gate

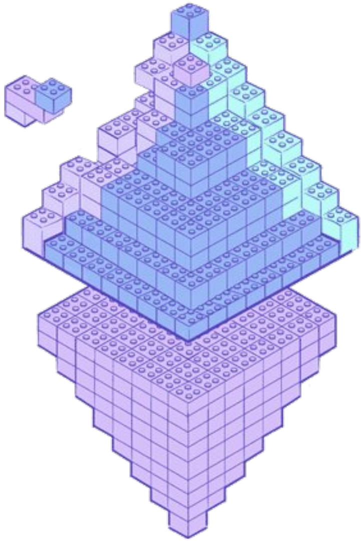
[100% Stmt, 100% Branch]

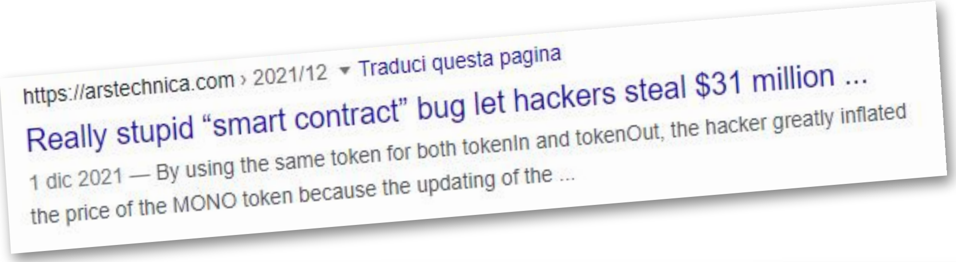
Deploy



```
-----------------|-----------|-----------|-----------|-----------|
File             |  % Stmts  |  % Branch |  % Funcs  |  % Lines  |
-----------------|-----------|-----------|-----------|-----------|
 contracts\      |      100  |      100  |      100  |      100  |
  CampusCoin.sol |      100  |      100  |      100  |      100  |
-----------------|-----------|-----------|-----------|-----------|
All files        |      100  |      100  |      100  |      100  |
-----------------|-----------|-----------|-----------|-----------|
```

Are we really done though?

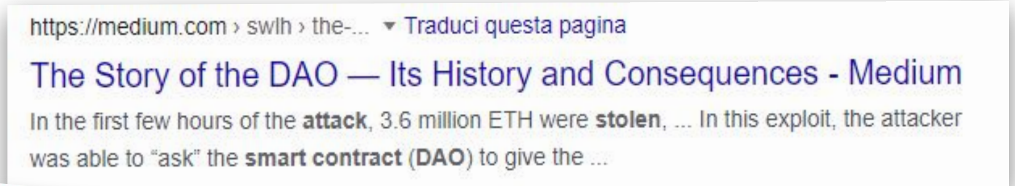




https://arstechnica.com › 2021/12 ▾ Traduci questa pagina

Really stupid "smart contract" bug let hackers steal $31 million ...

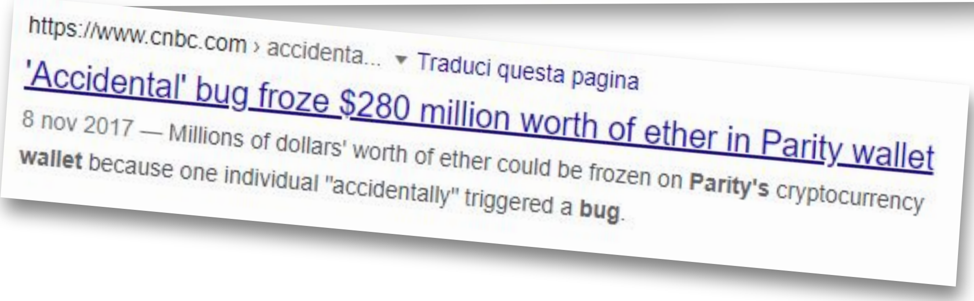
1 dic 2021 — By using the same token for both tokenIn and tokenOut, the hacker greatly inflated the price of the MONO token because the updating of the ...



https://medium.com › swlh › the-... ▾ Traduci questa pagina

The Story of the DAO — Its History and Consequences - Medium

In the first few hours of the **attack**, 3.6 million ETH were **stolen**, ... In this exploit, the attacker was able to "ask" the **smart contract** (**DAO**) to give the ...

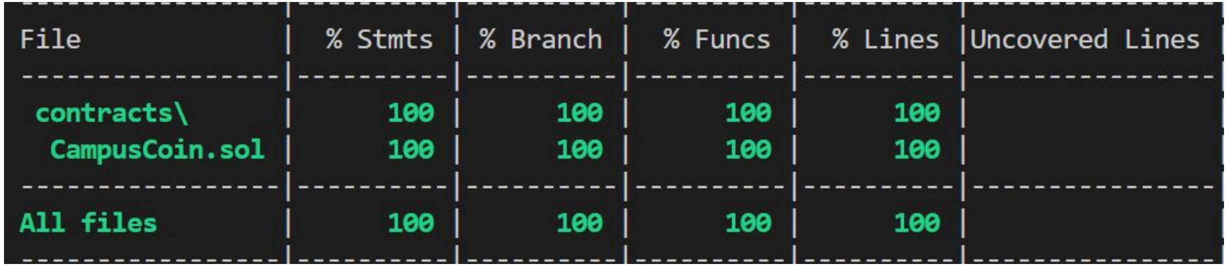


https://www.cnbc.com › accidenta... ▾ Traduci questa pagina

'Accidental' bug froze $280 million worth of ether in Parity wallet

8 nov 2017 — Millions of dollars' worth of ether could be frozen on **Parity's** cryptocurrency **wallet** because one individual "accidentally" triggered a **bug**.

Coverage Analysis: Results



```
-----------------|----------|----------|----------|----------|----------------
File             |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines
-----------------|----------|----------|----------|----------|----------------
 contracts\      |      100 |      100 |      100 |      100 |
  CampusCoin.sol |      100 |      100 |      100 |      100 |
-----------------|----------|----------|----------|----------|----------------
All files        |      100 |      100 |      100 |      100 |
-----------------|----------|----------|----------|----------|----------------
```

🛑 **Most Developers are inclined to stop here** 🛑

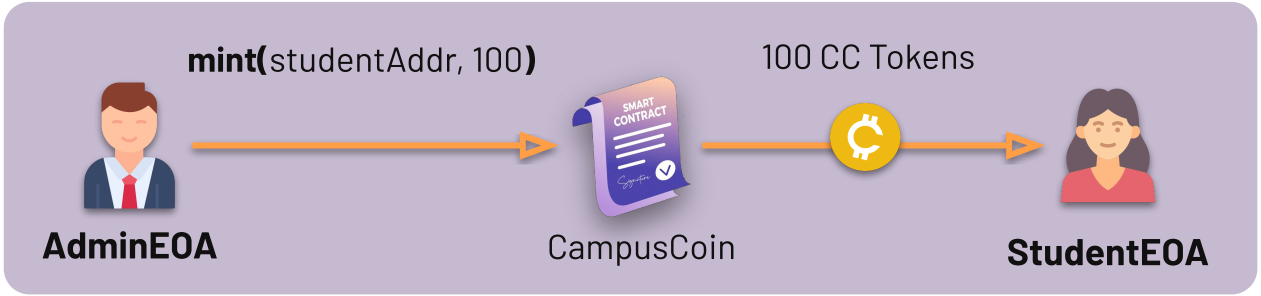
- **It’s a clear and familiar goal** → Intuitive and easily quantifiable.
- **It feels like “enough”** → Reaching 100% feels like a natural "done" signal.
- **It’s widely supported** → Well-integrated into most tools.

Coverage Analysis: Why it's not enough

1. **High coverage can give a false sense of security:**
   - It only shows that certain parts of the code were executed;
   - Not that they were meaningfully tested or properly verified.

2. **Coverage metrics are easy to game:**
   - Everyone can write tests that hit every line and branch;
   - We don't know if they're enforcing the right **expectations.**
   - It is possible to get **100% coverage** with **meaningless tests!**

Example: “Should mint tokens to a student”



CampusCoin implements a simple mint(address, amount) function:

- The Admin **mints** new CC tokens to a Student.
- The Student’s **token balance** is updated accordingly.

Example: “Should mint tokens to a student”

```
it("Should mint tokens to student", async function () {

      await campusCoin.mint(student1.address, "100");
      const studentBalance = await campusCoin.balanceOf(student1.address);

      expect(studentBalance).to.equal("100");
 });
```

CampusCoin.js: A **test method** verifies the correct behavior of mint().

1. **Simulate Tx:** Admin mints 100 new tokens to a Student;
2. **Assert:** Confirm the correctness of the program behavior:
   - Actual output: student balance
   - Expected output: 100 tokens

Example: “Should mint tokens to a student”

```
it("Should mint tokens to student", async function () {

      await campusCoin.mint(student1.address, "100");
      const studentBalance = await campusCoin.balanceOf(student1.address);

      //expect(studentBalance).to.equal("100");
 });
```

What happens if we remove the assertion?

- We are no longer enforcing any expectation;
- We expect test adequacy to decrease!

👉 **Let’s try it out:** **`npx hardhat coverage`**

Is Test Coverage Useless?

- **Good** for identifying under-tested parts of the system.
- **Bad** if used as a quality target!

— *Inozemtseva and Holmes (2014)*

Coverage Is Not Strongly Correlated with Test Suite Effectiveness

Laura Inozemtseva and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{lminozem,rtholmes}@uwaterloo.ca

ABSTRACT

The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size. In addition, most of the studies were done with adequate suites, which are are rare in practice, so the results may not generalize to typical test suites.

We have extended these studies by evaluating the relationship between test suite size, coverage, and effectiveness for large Java programs. Our study is the largest to date in the literature: we generated 31,000 test suites for five systems consisting of up to 724,000 lines of source code. We measured the statement coverage, decision coverage, and modified condition coverage of these suites and used mutation testing to evaluate their fault detection effectiveness.

We found that there is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for. In addition, we found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite. Our results suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics—*product metrics*

General Terms

Measurement

Keywords

Coverage, test suite effectiveness, test suite quality

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2756-5/14/05
http://dx.doi.org/10.1145/2568225.2568271

1. INTRODUCTION

Testing is an important part of producing high quality software, but its effectiveness depends on the quality of the test suite: some suites are better at detecting faults than others. Naturally, developers want their test suites to be good at exposing faults, necessitating a method for measuring the fault detection effectiveness of a test suite. Testing textbooks often recommend coverage as one of the metrics that can be used for this purpose (e.g., [29, 34]). This is intuitively appealing, since it is clear that a test suite cannot find bugs in code it never executes; it is also supported by studies that have found a relationship between code coverage and fault detection effectiveness [3, 6, 14–17, 24, 31, 39].

Unfortunately, these studies do not agree on the strength of the relationship between these test suite characteristics. In addition, three issues with the studies make it difficult to generalize their results. First, some of the studies did not control for the size of the suite. Since coverage is increased by adding code to existing test cases or by adding new test cases to the suite, the coverage of a test suite is correlated with its size. It is therefore not clear that coverage is related to effectiveness independently of the number of test cases in the suite. Second, all but one of the studies used small or synthetic programs, making it unclear that their results hold for the large programs typical of industry. Third, many of the studies only compared adequate suites; that is, suites that fully satisfied a particular coverage criterion. Since adequate test suites are rare in practice, the results of these studies may not generalize to more realistic test suites.

This paper presents a new study of the relationship between test suite size, coverage and effectiveness. We answer the following research questions for large Java programs:

Research Question 1. *Is the effectiveness of a test suite correlated with the number of test cases in the suite?*

Research Question 2. *Is the effectiveness of a test suite correlated with its statement coverage, decision coverage and/or modified condition coverage when the number of test cases in the suite is ignored?*

Research Question 3. *Is the effectiveness of a test suite correlated with its statement coverage, decision coverage and/or modified condition coverage when the number of test cases in the suite is held constant?*

The paper makes the following contributions:

- A comprehensive survey of previous studies that investigated the relationship between coverage and effectiveness (Section 2 and accompanying online material).

Mutation Testing

Mutation testing offers a stronger alternative, as it evaluates a test suite based on its ability to detect *small faults*.

- Tells us whether **assertions** are meaningful;
- If a **mutant survives**, the test suite **lacks a specific check** that can detect the fault;
- Encourages meaningful, bug-revealing tests.

SuMo - SOlidity MUtator

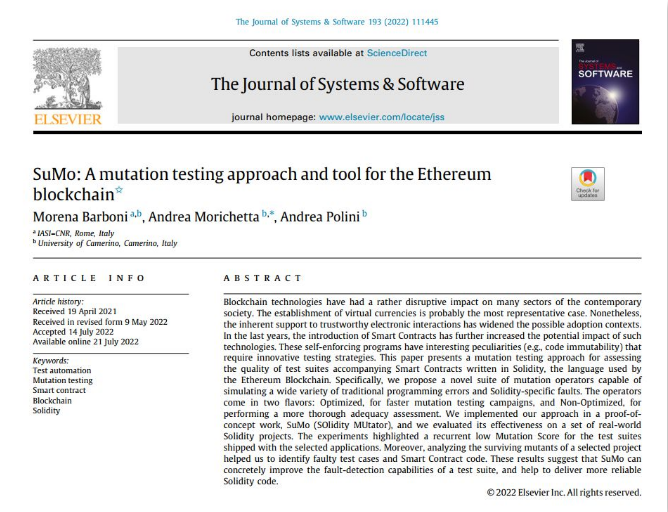
A flexible and **domain-aware** framework that comprehensively model **Solidity-specific faults** and guides the derivation of meaningful tests.

Automated Assessment

For any Solidity project regardless of frameworks.

40 Mutation Operators

- SOTA tools (PIT);
- Solidity constructs;
- Bugs and pitfalls.



The Journal of Systems & Software 193 (2022) 111445

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

ELSEVIER

SuMo: A mutation testing approach and tool for the Ethereum blockchain☆

Morena Barboni [a,b], Andrea Morichetta [b,*], Andrea Polini [b]

[a] IASI–CNR, Rome, Italy
[b] University of Camerino, Camerino, Italy

ARTICLE INFO

Article history:
Received 19 April 2021
Received in revised form 9 May 2022
Accepted 14 July 2022
Available online 21 July 2022

Keywords:
Test automation
Mutation testing
Smart contract
Blockchain
Solidity

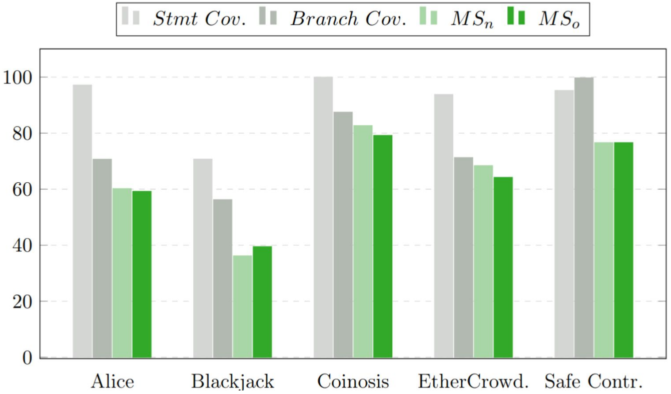
ABSTRACT

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. Nonetheless, the inherent support to trustworthy electronic interactions has widened the possible adoption contexts. In the last years, the introduction of Smart Contracts has further increased the potential impact of such technologies. These self-enforcing programs have interesting peculiarities (e.g., code immutability) that require innovative testing strategies. This paper presents a mutation testing approach for assessing the quality of test suites accompanying Smart Contracts written in Solidity, the language used by the Ethereum Blockchain. Specifically, we propose a novel suite of mutation operators capable of simulating a wide variety of traditional programming errors and Solidity-specific faults. The operators come in two flavors: Optimized, for faster mutation testing campaigns, and Non-Optimized, for performing a more thorough adequacy assessment. We implemented our approach in a proof-of-concept work, SuMo (SOlidity MUtator), and we evaluated its effectiveness on a set of real-world Solidity projects. The experiments highlighted a recurrent low 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.

Empirical Evaluation - Key Insights

Test Suites Achieved Low Quality Ratings

Average MS across projects = ~64%



Testers often overlook Solidity-specific constructs:

Traditional Mutants (MS = 68,2 %) | Solidity Mutants (MS = 61,2 %)

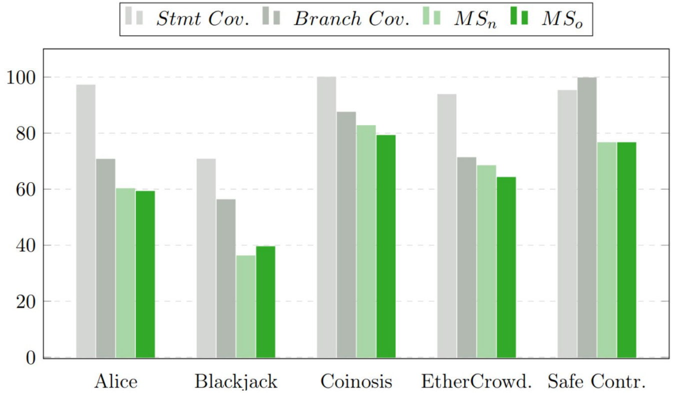
Target	MS	Potential Impact
Event	34%	Monitoring of contract behavior
Modifiers	37,7%	Access Control and reusable logic
Exception Handling	40,1%	Management of critical transaction reverting scenarios.
Blockchain Variable	64,5%	Any logic dependant on global blockchain properties (e.g., time)

SuMo can highlight such gaps in Test Suites that would otherwise remain overlooked.

Empirical Evaluation - Key Insights

Test Suites Achieved Low Quality Ratings

Average MS across projects = ~64%



Testers often overlook Solidity-specific constructs:

Traditional Mutants (MS = 68,2 %) | **Solidity Mutants (MS = 61,2 %)**

Target	MS	Potential Impact
Event	**34%**	Monitoring of contract behavior
Modifiers	**37,7%**	Access Control and reusable logic
Exception Handling	**40,1%**	Management of critical transaction reverting scenarios.
Blockchain Variable	**64,5%**	Any logic dependant on global blockchain properties (e.g., time)

SuMo can highlight such gaps in Test Suites that would otherwise remain overlooked.

Installing SuMo

You can install **SuMo** in two ways:

🔧 **Option 1: Direct Installation via NPM**

Run npm install @morenabarboni/sumo

📦 **Option 2: Add to package.json**

And then run npm install.

package/@morenabarboni/sumo

morenabarboni/sumo-solidity-mutator

SuMo-Config.js

It allows developers to customize how Sumo should behave within a project.

```
module.exports = {
      contractsDir: "auto",
      testDir: "auto",
      skipContracts: ["libraries"],
      skipTests: [],

      testingFramework: "auto",
      testingTimeOutInSec: 500,


      minimalOperators: false,
      randomSampling: false,
      randomMutants: 100
}
```

📁 Project Structure

Automatically detect or override contract, build, and test directories, and blacklists;

🧪 Test Execution

Configures test framework and timeout.

🧬 Mutation Strategy

Controls the scope and selection of mutants through minimal rules and random sampling.

Supported Testing Frameworks

SuMo is a **stand-alone module** that connects to the existing **testenv** to compile and test smart contracts, supporting broader project compatibility**.**

	HardHat	**Foundry**	**Brownie**	**Custom**
TEST LANGUAGE	JavaScript, TypeScript (mocha)	Solidity (forge)	Python (pytest)	Define Custom Test Script
CHAIN SIMULATOR	HardHat Network	Anvil	Ganache	–

Choosing Mutation Operators 👾

Category	ID	Mutation Example
Types, Units, and Locations	AVR, DLR, VUR	`someAddress` ⟶ `address(0)`
Function Modifiers	MOD, MOI, OMD, PKD	`function pay() payable` ⟶ `function pay()`
Global Variables and Functions	GVR, TOR	`tx.origin` ⟶ `msg.sender`
Return Semantics	RSD, RVS	`return transfer();` ⟶ `transfer(); return true;`
Math, Crypto and Libraries	MCR, SFR	`safeMath.add()` ⟶ `safeMath.sub()`

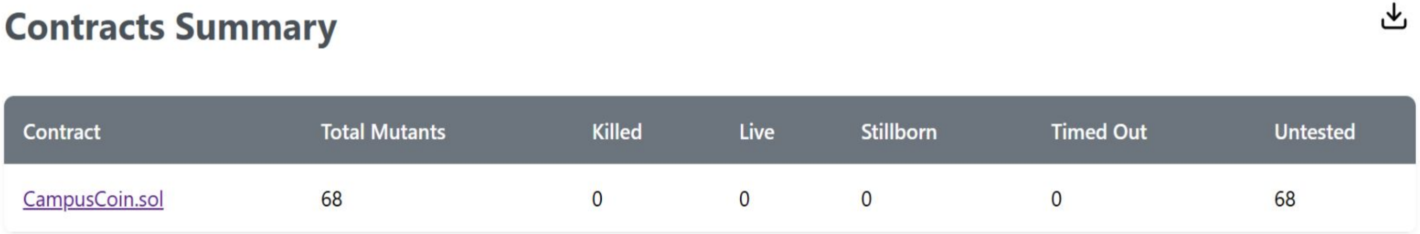
Operator Selection (Individual, Cluster)

- Useful for targeting specific aspects of smart contract behavior;
- e.g., arithmetic logic, state visibility, control flow, events.

Generating Mutants 👾

You can generate mutants without running tests: `npx sumo lookup`.

1. Parses Solidity source code into AST;
2. Explores nodes with each operator (custom visitor);
3. Applies rule-based mutations at matched nodes;
4. Generates a mutations.json file with all the mutants;



Contracts Summary

Contract	Total Mutants	Killed	Live	Stillborn	Timed Out	Untested
CampusCoin.sol	68	0	0	0	0	68

Mutant Pruning✂️

Focus testing efforts, manage runtime and analysis cost.

1) Minimal Operators

- Each operator injects one mutation per target;
- Rules empirically found to be effective.

2) Random Mutant Selection

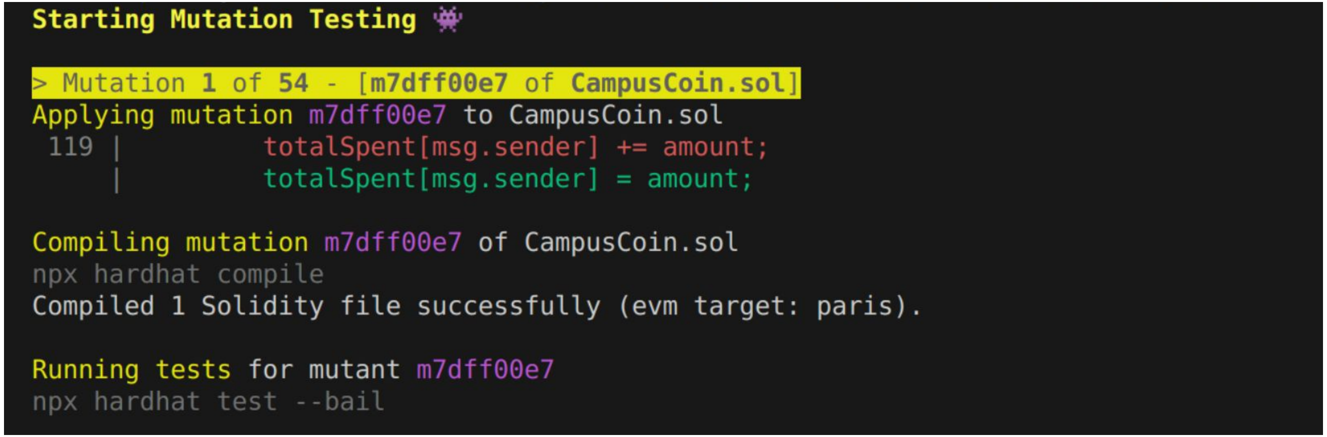
- Select a random subset of n mutants to be tested;

3) Coverage-Based Mutant Selection

- Target critical statements (with highest coverage);
- Target under-tested statements (with lowest coverage).

Running Mutation Testing ⏳

You can start mutation testing on the contracts under test by running `npx sumo test`. This actually starts the testing process:

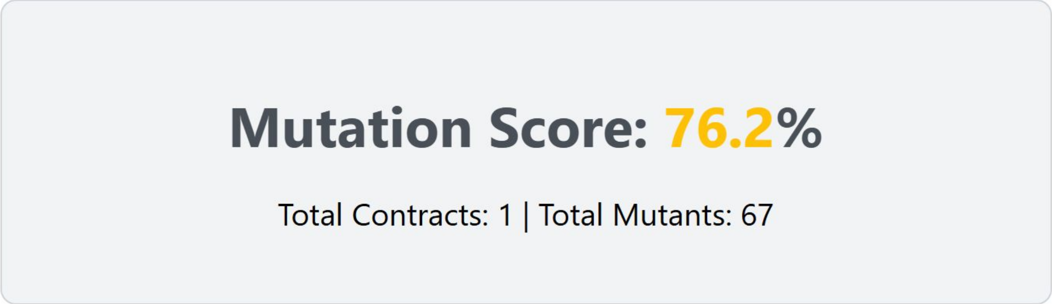


```
Starting Mutation Testing 👾

> Mutation 1 of 54 - [m7dff00e7 of CampusCoin.sol]
Applying mutation m7dff00e7 to CampusCoin.sol
 119 |         totalSpent[msg.sender] += amount;
     |         totalSpent[msg.sender] = amount;

Compiling mutation m7dff00e7 of CampusCoin.sol
npx hardhat compile
Compiled 1 Solidity file successfully (evm target: paris).

Running tests for mutant m7dff00e7
npx hardhat test --bail
```

□ Find the complete results in: SuMo-Demo/sumo-results

What can SuMo tell us about CampusCoin?

The Mutation Score is average despite achieving full coverage …



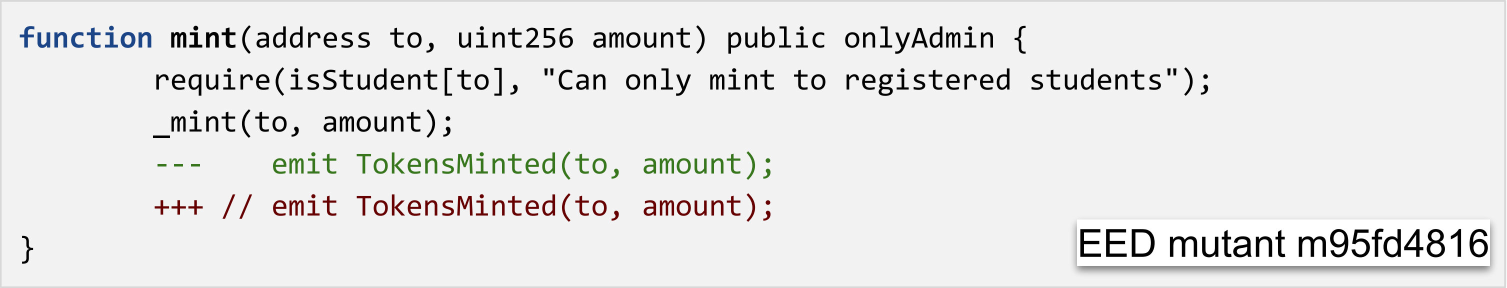
Mutation Score: 76.2%

Total Contracts: 1 | Total Mutants: 67

We can now **analyze live mutants** to derive **new tests** and improve the **fault-detection** of the CampusCoin.js test suite.

Mutation Analysis - Insight 1

All **EED (Event-Emission-Deletion)** mutants **survived** mutation testing.



```
function mint(address to, uint256 amount) public onlyAdmin {
        require(isStudent[to], "Can only mint to registered students");
        _mint(to, amount);
        ---    emit TokensMinted(to, amount);
        +++ // emit TokensMinted(to, amount);
}
```

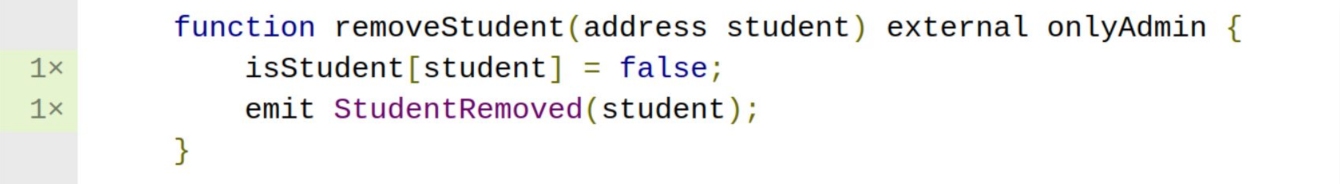
EED mutant m95fd4816

Implication: Our tests never check the correct emission of events!

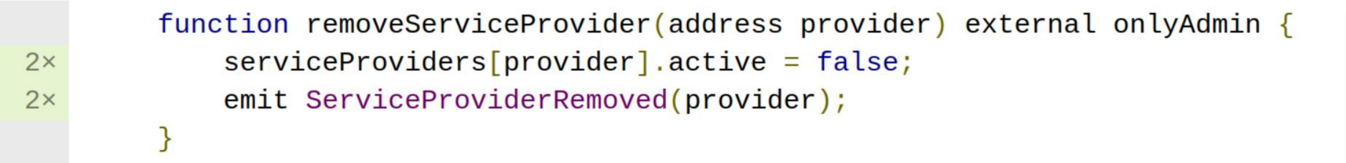
- **Events** → How contracts communicate with the outside world;
- **Faulty Events** → Off-chain systems may misinterpret key actions.

What about the Coverage Report?

In the Coverage Report all the **event emission statements** were covered at least once, giving us a false sense of security about their correctness.



```
        function removeStudent(address student) external onlyAdmin {
1×          isStudent[student] = false;
1×          emit StudentRemoved(student);
        }
```



```
        function removeServiceProvider(address provider) external onlyAdmin {
2×          serviceProviders[provider].active = false;
2×          emit ServiceProviderRemoved(provider);
        }
```

Updated Test to Kill EED Mutant

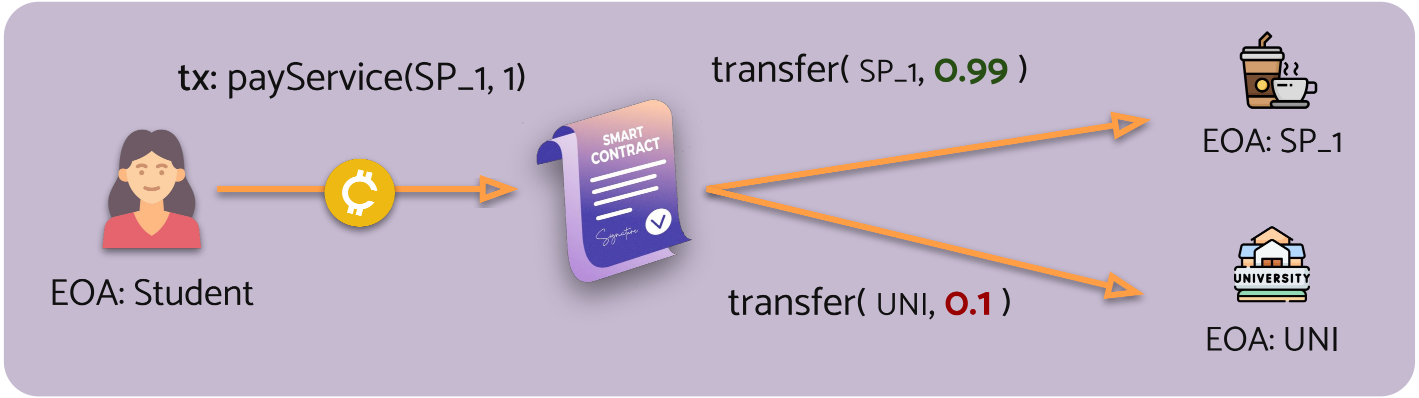
We used dedicated matchers to check for event emissions:

```
it("Should mint tokens to a student", async () {

   await expect(campusCoin.mint(student1.address,"100"))
      .to.emit(campusCoin, "TokensMinted")
      .withArgs(student1.address, "100");

   const balance = await campusCoin.balanceOf(student.address);
   expect(balance).to.equal(ethers.parseUnits("100", 18));
});
```

Test this with `npx sumo disable` □ `npx sumo enable EED` □ `npx sumo test`

□ Find the enhanced test suite in: solutions/3)Test-EED-Mutants-Killer.js

PayService() - Expected Behavior



A Student sends a payment to a Service Provider:

- 99% goes to the Service Provider;
- 1% fee is transferred to the University.

SuMo Report - Insight 2

Multiple **live mutants** around **fee-related logic** (both *fee computation* and *transfer*) suggest that this entire area of the contract is **under-tested**.

```
function paySerVice(address to, uint256 amount) external {
      ---|  _transfer(msg.sender, university, fee);
      +++| /* _transfer(msg.sender, university, fee); */
}
```

FCD mutant m88a18980

- This drastic mutation removes the fee transfer entirely!
- Yet, none of our tests detect the issue ...
- Let’s check the test method: “it: Should pay service”

SuMo Report - Insight 2

The enhanced test case exposed a bug in the original Smart Contract!

```
function payService(address to, uint256 amount) external {
    uint256 fee = (amount / 100) * UNIT; // fee computation
}
```

Precision Loss Bug: Amount first divided by 100 then multiplied by UNIT:

● Test Input:	amount =1 token;	
● Expected Fee:	1 / 100 = **0.01 tokens**	//cannot represent with uint256
● Actual Fee:	fee = 0 tokens	//drops the fractional part

SuMo Report - Insight 2

The enhanced test case exposed a bug in the original Smart Contract!

```
function payService(address to, uint256 amount) external {
    uint256 fee = (amount / 100) * UNIT; // 1%
}
```

BUGGY CONTRACT

```
function payService(address to, uint256 amount) external {
    uint256 fee = (amount * UNIT) / 100; // 1%
}
```

FIXED CONTRACT

Conclusions

Mutation testing is a powerful complement to coverage analysis - it doesn’t tell you ***where your tests go***, but ***what they actually prove***.

- Gives a more meaningful measure of test suite quality;
- Reduce the risk of undetected bugs before deployment.

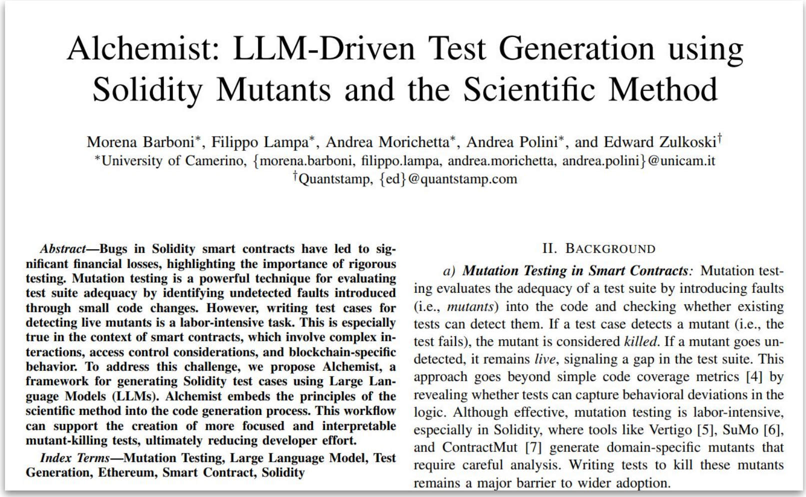
There’s no such thing as a free lunch:

- Running Mutation Testing is time consuming;
- Analyzing mutants can be overwhelming.

Mutant-Driven Test Generation Via LLMs

Alchemist automatically improves the **quality** of existing **unit tests**:

- Identifies **quality gaps** using **Mutation Testing**;
- Automatically **fills these quality gaps** using **LLMs**.



Alchemist: LLM-Driven Test Generation using Solidity Mutants and the Scientific Method

Morena Barboni*, Filippo Lampa*, Andrea Morichetta*, Andrea Polini*, and Edward Zulkoski†
*University of Camerino, {morena.barboni, filippo.lampa, andrea.morichetta, andrea.polini}@unicam.it
†Quantstamp, {ed}@quantstamp.com

***Abstract*—Bugs in Solidity smart contracts have led to significant financial losses, highlighting the importance of rigorous testing. Mutation testing is a powerful technique for evaluating test suite adequacy by identifying undetected faults introduced through small code changes. However, writing test cases for detecting live mutants is a labor-intensive task. This is especially true in the context of smart contracts, which involve complex interactions, access control considerations, and blockchain-specific behavior. To address this challenge, we propose Alchemist, a framework for generating Solidity test cases using Large Language Models (LLMs). Alchemist embeds the principles of the scientific method into the code generation process. This workflow can support the creation of more focused and interpretable mutant-killing tests, ultimately reducing developer effort.**

***Index Terms*—Mutation Testing, Large Language Model, Test Generation, Ethereum, Smart Contract, Solidity**

II. Background

a) Mutation Testing in Smart Contracts: Mutation testing evaluates the adequacy of a test suite by introducing faults (i.e., *mutants*) into the code and checking whether existing tests can detect them. If a test case detects a mutant (i.e., the test fails), the mutant is considered *killed*. If a mutant goes undetected, it remains *live*, signaling a gap in the test suite. This approach goes beyond simple code coverage metrics [4] by revealing whether tests can capture behavioral deviations in the logic. Although effective, mutation testing is labor-intensive, especially in Solidity, where tools like Vertigo [5], SuMo [6], and ContractMut [7] generate domain-specific mutants that require careful analysis. Writing tests to kill these mutants remains a major barrier to wider adoption.

Thursday, June 5 17:07 - S3.2:ShortPapers - Room: *PS1*