Automated Vulnerability Injection in Solidity Smart Contracts: A Mutation-Based Approach for Benchmark Development

Gerardo Iuliano, Luigi Allocca, Matteo Cicalese, Dario Di Nucci
University of Salerno
Fisciano (SA), Italy
geiuliano@unisa.it,l.allocca8@studenti.unisa.it,mcicalese@unisa.it,ddinucci@unisa.it

Abstract

The security of smart contracts is critical in blockchain systems, where even minor vulnerabilities can lead to substantial financial losses. Researchers proposed several vulnerability detection tools evaluated using existing benchmarks. However, most benchmarks are outdated and focus on a narrow set of vulnerabilities. This work evaluates whether mutation seeding can effectively inject vulnerabilities into Solidity-based smart contracts and whether state-of-the-art static analysis tools can detect the injected flaws. We aim to automatically inject vulnerabilities into smart contracts to generate large and wide benchmarks. We propose *MuSe*, a tool to generate vulnerable smart contracts by leveraging pattern-based mutation operators to inject six vulnerability types into real-world smart contracts. We analyzed these vulnerable smart contracts using Slither, a static analysis tool, to determine its capacity to identify them and assess their validity. The results show that each vulnerability has a different injection rate. Not all smart contracts can exhibit some vulnerabilities because they lack the prerequisites for injection. Furthermore, static analysis tools fail to detect all vulnerabilities injected using pattern-based mutations, underscoring the need for enhancements in static analyzers and demonstrating that benchmarks generated by mutation seeding tools can improve the evaluation of detection tools.

CCS Concepts

• **Software and its engineering → Software verification and validation**; • **Security and privacy → Software security engineering**.

Keywords

Vulnerability, Smart Contract, Mutation Testing, Benchmark

ACM Reference Format:
Gerardo Iuliano, Luigi Allocca, Matteo Cicalese, Dario Di Nucci. 2018. Automated Vulnerability Injection in Solidity Smart Contracts: A Mutation-Based Approach for Benchmark Development. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

EASE 2025, Istanbul, Türkiye

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Smart contracts are self-executing programs operating on blockchain platforms like Ethereum [7]. They enable decentralized applications to execute pre-defined terms and conditions autonomously, promoting trust between untrusted parties. With the advancement of blockchain technology, smart contracts have become indispensable in various domains, including digital payments and decentralized finance [35]. However, their immutable nature, while foundational to the blockchain trust model, presents significant challenges when vulnerabilities are discovered [26]. Unlike traditional software, deployed smart contracts cannot be modified or patched, making even minor flaws potentially catastrophic [1].

The security of smart contracts is critical. Vulnerabilities such as *Reentrancy*, *Timestamp dependence*, *Transaction Order Dependence* (TOD), *Authorization through tx.origin*, and *Unchecked external calls* have already led to significant financial losses, exemplified by the infamous DAO hack [27]. Researchers and developers have created a myriad of vulnerability detection tools. Often leveraging static or dynamic analysis, these tools aim to identify and mitigate potential defects before deployment and, in some cases, on-chain. Despite their advancements, these tools are not infallible. False positives [30], false negatives, and the inability to detect complex vulnerability patterns limit their effectiveness, underscoring the need for robust evaluation.

One critical barrier to improving detection tools is the lack of comprehensive and diverse benchmarks [13]. Existing datasets are often outdated, limited in size, or narrowly focused on specific vulnerabilities, leaving many tools untested against realistic scenarios. The most commonly used benchmarks [15] comprise smart contracts written using the old Solidity version and affected by a subset of the known vulnerability. Other datasets often contain simplistic toy contracts [39], like the SWC Registry. In addition, about 96% of smart contracts present in datasets are involved in no more than five transactions [9]. Ren *et al.* [32] highlighted that tools should be evaluated using a comprehensive benchmark suite that integrates multiple vulnerability types. The absence of high-quality and updated benchmarks hinders progress in the field by affecting the ability to validate and improve detection tools.

To address this gap, we propose *MuSe*, a mutation-based tool based on SuMo [2] to generate benchmarks by injecting vulnerabilities into smart contracts. By leveraging mutation operators designed around known vulnerability patterns, our approach systematically introduces faults into realistic scenarios, enabling the generation of contracts with vulnerabilities placed in both typical and unconventional yet valid locations, challenging detection tools to expand their scope and improve accuracy. Such versatility allows for evaluating detection tools against various scenarios,

including edge cases. The strength of our approach lies in its foundation on pattern-based mutation operators. These operators are designed to inject vulnerabilities wherever their corresponding patterns are identified, ensuring consistency and extensibility; as new vulnerabilities are identified, corresponding operators can be added with minimal effort. We manually validated mutation operators and evaluated the generated benchmarks using Slither [16], a state-of-the-art static analysis tool for smart contracts.

The results reveal that each vulnerability has a different injection rate, and successful injection depends on whether the smart contract satisfies the necessary preconditions to adhere to vulnerability patterns. Detection outcomes reveal significant limitations of Slither in identifying vulnerabilities, especially when they are injected into unconventional or unexpected locations. These findings underscore the weaknesses of current static analysis tools and highlight the pressing need for more advanced detection techniques. Moreover, *MuSe* allows to increase the benchmark size of above 840%. In conclusion, our study identifies the gaps in the static analyzer, offering a tangible solution for researchers and practitioners to enhance or generate new benchmarks to evaluate their detection tools. Our work provides the following contributions:

- *MuSe*, a mutation seeding tool to generate benchmarks injecting vulnerabilities in Solidity smart contracts using pattern-based mutation operators;
- an enhanced version of the dataset smartbugs-wild containing 350,493 vulnerable smart contracts;
- a list of weaknesses that affect the capabilities of Slither in detecting vulnerabilities.

Paper Structure. This paper is organized as follows. Section 2 establishes the background and reviews related work. Section 3 details the research method, including research questions and the mutation-based approach. Section 4 presents the experimental results, while Section 5 discusses key findings, implications, and their relevance. Section 6 addresses threats to validity, and Section 7 concludes the paper and provides future research directions.

2 Background and Related Work

This section introduces smart contracts, their vulnerabilities, and mutation testing. Furthermore, we provide some related work that is relevant to our study.

2.1 Smart Contracts

Smart contracts are self-executing programs designed to automatically enforce terms and conditions between untrusted parties [23]. Initially envisioned as a way to automate legal contracts, the rise of blockchain technology has transformed smart contracts into scripts that execute synchronously across nodes in a distributed ledger [40]. On the Ethereum blockchain, smart contracts run within the Ethereum Virtual Machine (EVM), a Turing-complete, stack-based virtual machine that ensures isolated contract code execution. A smart contract is identified by a unique address, private storage, and a balance in Ether, and it contains executable code. When a transaction is sent to a contract address, it triggers the contract functionality, providing invocation data and paying transaction fees using Gas [23].

2.2 Smart Contract Vulnerabilities

Ethereum smart contracts are prone to various vulnerabilities unique to blockchain technology. Reentrancy is one of the most critical [24, 25], as evidenced by the infamous DAO hack, where an attacker repeatedly called back into a contract to drain its funds. Another major issue is the misuse of transaction origin for authorization, which attackers can easily spoof to gain unauthorized access. Timestamp manipulation by miners is another concern, allowing them to alter timestamps and compromise the security of critical contract functions [22, 28]. Transaction-ordering dependence (TOD), where the order of transactions is unpredictable, can be exploited to unfairly manipulate outcomes, such as reducing rewards before submitting a valid solution [14, 34, 37]. External calls also pose significant risks. Attackers can exploit these calls to execute malicious code. Furthermore, failing to handle a function return value properly may enable attackers to drain contract balances. Denial of Service (DoS) attacks can arise in several ways [33], such as through costly external calls or inefficient looping behavior. Smart contract development differs fundamentally from traditional software programming. Vulnerabilities in smart contracts deployed on public blockchains are particularly challenging to fix due to the immutable nature of blockchain systems. While some traditional security techniques are applicable, smart contracts introduce unique challenges, and many vulnerabilities arise from the distinctive characteristics of blockchain technology.

2.3 Mutation Testing

Mutation testing is a software testing technique to evaluate the effectiveness of test cases by intentionally introducing small changes, called mutants, into the source code to simulate potential faults or errors. The primary goal is to assess whether the existing test cases can detect these changes, thereby measuring the fault-detection capability of a test suite [21, 29]. This process ensures software is rigorously validated, improving its reliability and reducing the likelihood of undetected faults in production.

Mutation testing in Solidity applies the same principles as traditional mutation testing. Still, it focuses on the unique characteristics of smart contracts, injecting Solidity-specific faults to evaluate the effectiveness of vulnerability detection and test suite robustness. In the literature, some tools have been proposed for this purpose. Chapman *et al.* proposed Deviant [8], a mutation testing tool designed for Solidity smart contracts. It automatically generates mutated versions of a given Solidity project and runs them against existing test suites to assess their effectiveness. Deviant includes mutation operators that cover Solidity-specific features based on a Solidity fault taxonomy and traditional programming constructs to simulate faults. The authors used Deviant to evaluate the test effectiveness of three Solidity projects. Their findings show that achieving high statement and branch coverage in Solidity does not guarantee strong code quality. This study provides valuable insights for Solidity developers, emphasizing the need for more rigorous testing to minimize financial risks. Ivanova and Khritankov presented RegularMutator [20], a tool for improving the reliability of smart contracts written using Solidity language. RegularMutator implements language-specific operators that correspond to common errors made during the development of smart contracts.

Injection of mutations in program code is implemented using regular expressions. The tool demonstrated its effectiveness in testing large-scale smart contract projects. The study concluded that mutation analysis provides a more reliable measure of test suite quality than traditional test line coverage. Barboni *et al.* proposed SuMo [3], a mutation testing tool designed for Solidity smart contracts, incorporating 25 Solidity-specific mutation operators alongside 19 traditional ones. It enables mutation testing on Solidity projects to assess test effectiveness. SuMo was later extended by ReSuMo [4], which introduces a regression mutation testing approach. ReSuMo employs a static, file-level technique to selectively mutate a subset of smart contracts and rerun only relevant test cases during regression testing. After each mutation testing run, ReSuMo incrementally updates its results by leveraging test outcomes from previous program revisions, improving efficiency and reducing redundant computations.

2.4 Related Work

Bug injection is a testing method widely studied in traditional software programs; however, only a few studies have addressed its application to smart contracts using mutation security testing.

Ghaleb and Pattabiraman proposed *SolidiFI* [17], an automated and systematic approach to evaluate static analysis tools. The tool injects bugs into a smart contract to introduce the targeted vulnerabilities and then checks the generated buggy contracts using the static analysis tools. The tool injects bugs by adding vulnerable code snippets, code transformations, and weakening security mechanisms. We extended the code transformation approach, focusing on more vulnerabilities and a larger dataset to experiment. We also validated the mutation seeding tool to provide a tool to generate new vulnerable smart contracts starting from an initial set.

Chu *et al.* [10] introduced *SGDL* (Smart Contract Vulnerability Generation with Deep Learning), an approach to create authentic and diverse vulnerability datasets for smart contracts. *SGDL* combines generative adversarial networks (GANs) with static analysis to extract syntactic and semantic information from contracts. Using this information, it generates realistic vulnerability fragments and injects them into smart contracts via an abstract syntax tree, ensuring syntactic correctness. The approach depends on a labeled dataset and researchers' expertise in vulnerabilities to train the GANs. However, comprehensive datasets for various vulnerabilities remain scarce or are not openly accessible for academic research. Consequently, *SGDL* focus is limited to specific vulnerabilities. In contrast, our mutation-based approaches rely on predefined patterns and manually designed mutation operators. New operators do not depend on datasets but on vulnerability patterns, making the approach easy to extend to other vulnerabilities.

Hajdu *et al.* [18] conducted a study using software-implemented fault injection (*SWIFI*) to evaluate the dependability of permissioned blockchain systems in the presence of faulty smart contracts. They introduced general software and blockchain-specific faults into smart contract code to assess their impact on system reliability and integrity. They also investigated the effectiveness of formal verification and runtime protection mechanisms in detecting and mitigating these faults. The authors used Hyperledger Fabric and evaluated 15 smart contracts, each tested in three versions: a base version, a version with extensive protections, and a version without protections. Faults were injected into the unprotected versions, resulting in 651 faulty variants. The findings revealed that formal verification and runtime protections complement built-in platform checks but cannot detect all faults. Our study focused on a significantly larger dataset of faulty smart contracts and targeted critical smart contract vulnerabilities. Additionally, the goal was distinct, focusing on generating benchmarks that can be used to validate and improve the effectiveness of vulnerability detection tools.

Regarding benchmark generation, other techniques have been explored in the literature, such as analyzing audit reports and leveraging large language models (LLMs). On the one hand, Zheng *et al.* [39] created a large-scale dataset of SWC weaknesses from real-world DApp projects. They recruited 22 participants to analyze 1,199 open-source audit reports from 29 security teams, identifying 9,154 weaknesses. Their work resulted in two distinct datasets. The DAppSCAN-Source dataset contains 39,904 Solidity files with 1,618 SWC weaknesses. However, these files may not be directly compilable for automated analysis. To address this, the authors developed a tool that automatically identifies dependency relationships within DApp projects and resolves missing public libraries. The DAppSCAN-Bytecode dataset includes 6,665 compiled smart contracts containing 888 SWC weaknesses. Evaluation results showed that existing detection tools perform poorly on these datasets, highlighting the need for future research to focus on real-world smart contract datasets rather than simplistic toy contracts. On the other hand, Daspe *et al.* [11] utilized large language models (LLMs) to generate a dataset of Solidity smart contracts. To guide the LLM during the generation process, they adopted an approach inspired by Test-Driven Development (TDD) [5]. Each prompt was submitted to an LLM, producing Solidity code that was then parsed. The Solidity compiler verified the syntax, and if the contract compiled successfully, Slither was applied for static analysis to detect vulnerabilities. Next, they created a project containing the generated contract and functional tests, which were then executed. They collected the compilation status, vulnerability report, and functional test results for each prompt. After multiple generations, they conducted an evaluation based on prompt complexity and the model used. The study found that LLMs struggled with increased complexity, demonstrating low accuracy as contract intricacy grew. Most compilation errors arose from incorrect type usage, like strings and arrays, and issues related to the payable/call pattern.

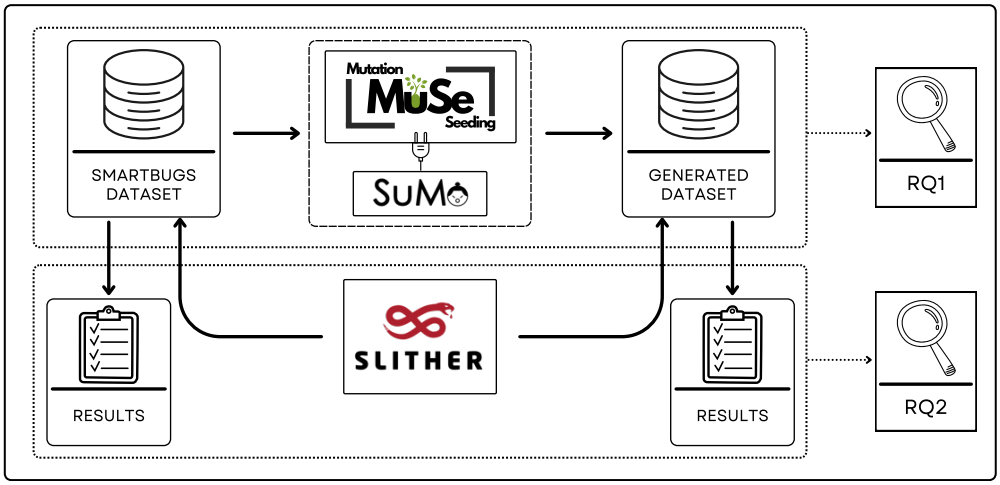
The work described above highlights the challenges faced and the different approaches used in literature to generate benchmarks. Our work bridges some of the gaps by offering a possible solution.

3 Research Method

The following section presents the details of the study, highlighting the main goal and its related research questions.

◎ Goal of the study.
Our goal is to automatically inject vulnerability in smart contracts to generate large and wide benchmarks that researchers and developers can use to improve detection tool evaluation.

Figure 1: Summary of the Research Method.



To achieve this goal, we developed *MuSe*, a tool to generate vulnerable smart contracts that are challenging to detect automatically. Inspired by mutation testing [21, 29], we extended the SuMo mutation testing tool [2] to mutate Solidity smart contracts into vulnerable mutants. To this end, we implemented mutation operators designed to inject known vulnerabilities by modifying the smart contracts appropriately. Afterward, we analyzed to what extent the injected vulnerabilities are challenging to detect by observing how well static analyzers can detect them. We leveraged Slither [16], a popular static analyzer to detect vulnerabilities in Solidity, and compared the results achieved by running it before and after the mutation phase, assessing the presence of the injected vulnerabilities and the performance of Slither in detecting them.

Our motivation is the critical role that smart contract security plays in the blockchain and the limitations of existing vulnerability detection tools in handling complex vulnerabilities. By combining mutation security testing with static analysis, this work seeks to provide empirical evidence of the strengths and weaknesses of tools like Slither, offering insights that can drive the development of more robust security solutions and methodologies for smart contracts. Figure 1 depicts our research method.

As shown in Table 1, we selected six vulnerabilities [19, 31, 36] to inject based on their relevance in the literature and the ability of Slither to detect them with at least medium confidence: *Unchecked call return value*, *Unchecked send*, *Authentication through tx.origin*, *Delegatecall to untrusted callee*, and *Unused return*. These vulnerabilities are among the top 15 most discussed in the literature, as highlighted in the work of Zaazaa Oualid and El Bakkali Hanan [38]. Based on our goal, we formulate these research questions (RQs):

> **RQ$_1$.** *To what extent are the mutation operators implemented in MuSe generalizable?*

RQ_1 allows us to assess the feasibility of introducing vulnerabilities into real-world smart contracts and understand the generalizability of the mutation operators to inject a vulnerability.

> **RQ$_2$.** *How can the mutants injected by MuSe be detected through static analysis?*

RQ_2 analyzes the performance of static analyzer when detecting vulnerabilities injected into contracts. The goal of applying various patterns to introduce vulnerabilities is twofold. On the one hand, we aim to identify whether the injected vulnerabilities are challenging to detect. On the other hand, we want to evaluate whether a static analysis tool can identify these specific patterns, highlighting its strengths and weaknesses.

3.1 Data Collection

To answer our research questions, we used the real-world dataset smartbugs-wild[1], which contains 47,398 smart contracts extracted from the Ethereum network that have at least one transaction. We ran Slither using SmartBugs [12], which allows us to parallelize its execution using several Docker images and easily parse the results.

3.2 Mutation Operators

We extended SuMo [2], a mutation testing tool, by creating new mutation operators focusing on security. The tool uses the *solidity-parser-antlr*[2], a parser built from a robust ANTLR4 grammar, which generates an Abstract Syntax Tree of the code based on the Solidity grammar. We implement a mutation operator for each vulnerability to inject and leverage the parser to identify injection patterns where the vulnerabilities can be injected. Table 1 maps the implemented mutation operators, the vulnerability they inject, and the detector Slither uses to identify them.

UC Operator. To inject the *Unchecked low-level call return value*, we identify the possible instructions or statements to mutate. We identified all the low-level call functions. We mutated the contract by removing the controls whenever the return value of the functions was checked using the require function or an if statement.

[1] https://github.com/smartbugs/smartbugs-wild
[2] https://github.com/solidity-parser/parser

Table 1: Vulnerabilities injected into smart contracts by *MuSe*.

Vulnerability	Description	Slither's Detector	Operator
Unchecked low-level call return value	Low-level calls return *false* on failure instead of throwing exceptions, risking critical vulnerabilities if unchecked.	`unchecked-lowlevel`	UC
Unchecked send	The send function returns *false* on failure without throwing an exception, risking vulnerabilities if unchecked.	`unchecked-send`	US
Authentication via tx.origin	Using *tx.origin* for authorization risks vulnerabilities if an authorized account interacts with a malicious contract.	`tx-origin`	TX
Unused return	The return value of an external call is not stored in a local or state variable.	`unused-return`	UR
Multiple calls in a loop	Calls inside a loop might lead to a denial-of-service attack.	`calls-loop`	CL
Delegatecall to untrusted callee	*Delegatecall* executes the code at the target address in the context of the calling contract. It allows a SC to load code dynamically from a different address.	`controlled-delegatecall`	DTU

```
// Before UC mutation
function withdraw(uint amount) public {
  require(msg.sender.call.value(amount)());
}
// After UC mutation
function withdraw(uint amount) public {
  msg.sender.call.value(amount)();
}
```

US Operator. To inject *Unchecked send*, we identified all the send functions and removed any control on the return value.

```
// Before US mutation
function sendEth(address payable giftee) public {
  if (!giftee.send(1 ether)) {
    revert("Send failed");
  }
}
// After US mutation
function sendEth(address payable giftee) public {
  giftee.send(1 ether)
}
```

TX Operator. Authentication through tx.origin was injected substituting the `msg.sender` variable with `tx.origin`. We mutate the contracts when the `msg.sender` variable is used in a binary operation like "==" to check the ownership of the contract or a specific address having some privileges or access to the asset.

```
// Before TX mutation
modifier onlyOwner() {
  require(msg.sender == owner, "No owner"); _;
}
// After TX mutation
modifier onlyOwner() {
  require(tx.origin == owner, "No owner"); _;
}
```

UR Operator. Unused return was injected in two cases. First, when a binary operation like "=" assigns the return value of a function to a variable. Second, when a variable is declared and then initialized with the return value of a function. In both cases, we removed the left side of the assignment and left the call function.

```
// Before UR mutation
function addNumbers(uint256 a, uint256 b) public {
  c = SafeMath.add(a, b);
}
// After UR mutation
function addNumbers(uint256 a, uint256 b) public {
  SafeMath.add(a, b);
}
```

CL Operator. Multiple calls in a loop were injected by wrapping call, send, and transfer functions in a statement of 1,000 loops implemented using the "`for`" construct.

```
// Before CL mutation
function payMember(address payable member) public {
  require(member.send(0.1 ether);
}
// After CL mutation
function payMember(address payable member) public {
  for (uint256 i = 1; i <= 5; i++) {
    require(member.send(0.1 ether);
  }
}
```

DTU Operator. Delegatecall to untrusted callee was injected by introducing a new address variable and a new function that allows users to replace the address variable with another. Each use of `delegatecall` was mutated, replacing the address used to delegate with the new personalizable address, which can be set to malicious.

```
// Before DTU mutation
function setFalseValue(address _address) public {
  require(_address.delegatecall(
    abi.encodeWithSignature("setFalse(uint256)")));
}
// After DTU mutation
address public delegate;
function setDelegate(address _delegate) public {
  delegate = _delegate;
}
function setFalseValue(address _address) public {
  require(delegate.delegatecall(
    abi.encodeWithSignature("setFalse(uint256)")));
}
```

3.3 *MuSe* Validation

To ensure the validity of our results, we manually validated *MuSe* on a statistically significant subset of mutated smart contracts with a 95% confidence level and a 5% margin error. The tool generated

350,716 mutated contracts, from which we randomly selected a subset of 384 smart contracts, representing a statistically significant sample size. Our validation process involved manually analyzing each contract to verify whether SuMo injected the intended vulnerabilities correctly. The procedure consisted of these steps:

(1) *Compilation for Syntactical Correctness.* Each selected smart contract was compiled to ensure its syntactical correctness.
(2) *Comparison with SuMo Logs.* We compared the logs provided by SuMo, which detail the applied mutation type and the lines of code affected, against the corresponding mutated smart contracts.
(3) *Pattern Adherence Verification.* We verified that the points where vulnerabilities were injected adhered to the patterns defined by the mutational operator. This step ensured that SuMo identified the correct lines of code and statements for injecting vulnerabilities.
(4) *Modification Assessment.* We examined SuMo's modifications or additions to the original code to determine whether the injected vulnerabilities were accurately implemented.

A mutation was marked as correctly injected if (i) the mutated contract contained the new lines of code introducing the target vulnerability or (ii) existing lines of code were altered to render the smart contract vulnerable to the intended issue.

The validation results show that our tool failed to inject vulnerabilities in 20 out of 384 smart contracts, or 5.21% of cases. The main causes of these failures are exceptional cases that the mutational operator does not adequately handle. One common issue arises when the mutated statement contains a semicolon (";"). Strings including a semicolon within the mutated statement can interfere with the operator logic, leading to unintended code truncation. As a result, the generated mutant may have incorrect syntax, rendering the contract uncompilable. Another cause of injection failure is conflicts between the scopes of contract variables and the variables introduced by the mutation. For example, the CL operator injects a for loop that uses the variable `uint i` for iteration. If the mutation is applied to a statement declaring a `uint i` variable, the compiler cannot differentiate between the two variables, leading to a compilation error. In all other cases, the mutation process successfully injects the vulnerability without issues, producing valid mutants.

3.4 Replication Package

We have made *MuSe* publicly available on GitHub[3], allowing researchers and practitioners to replicate our study or utilize the tool for their purposes. Additionally, we have uploaded the sample used to manually validate the mutation operators.[4]

4 Empirical Results

We ran *MuSe* on the SmartBugs-wild dataset, applying the six previously described mutation operators to each contract. Starting from 47,398 smart contracts, we generated 350,493 vulnerable ones. *MuSe* mutates a contract each time it matches the pattern of a mutation operator. A contract could exhibit more than a pattern. Table 2 shows the number of mutants generated by each operator and highlights the number of contracts suitable to be mutated by each operator. *MuSe* mutated 41,337 out of 47,398 smart contracts, about 87% of cases. The remaining 6,061 smart contracts were not mutated for two reasons: (i) the absence of any patterns used by mutation operators in 5,990 smart contracts and (ii) the invalid content of the files for 71 of them, e.g., a JSON representation of the smart contract instead of well-formatted Solidity code.

4.1 RQ1. To what extent are the mutation operators implemented in *MuSe* generalizable?

To answer RQ_1, we observed the number of smart contracts that could be correctly mutated. We aimed to understand how many pattern occurrences applied by each mutation operator could be injected in real-world scenarios.

Table 2: Results for mutation operators ordered by injection rate, total number of generated mutants, and average injection rate of *MuSe*.

Operator	# Mutated SCs	# Mutants	Injection Rate
UR	33,910	213,912	71.50%
TX	32,250	65,825	68.00%
CL	26,604	61,687	56.00%
UC	4,094	4,992	8.60%
US	2,248	3,928	4.70%
DTU	113	149	0.23%
-	-	350,493	34.83%

As shown in Table 2, the most injectable vulnerability is the *unused return*, with an injection rate of 71.5%. The patterns used to inject vulnerabilities are not only common but also frequently encountered in smart contracts, highlighting their prevalence in typical contract design. On average, each smart contract contains six occurrences of these patterns, reflecting their foundational role in contract development. Patterns such as assignments, declarations, and initializations are essential building blocks in smart contract programming. However, their prevalence also increases the likelihood of vulnerabilities arising from improper or unintended usage.

The second vulnerability, *authorization via tx.origin*, is injectable in a real-world scenario in 68% of cases. The high injection rate respects the frequency of the pattern of the TX operator in the smart contracts. As described in Section 3, the TX operator mutates a contract when the variable `msg.sender` is used to check the ownership of a contract or the privilege of an address on the asset.

The injection rate for *multiple calls in a loop* is 56%, showing only just over half of the contracts involve the use of a `call`, `send`, or `transfer` function. The moderate injection rate highlights that these functions are commonly employed in contracts but not excessively frequent. In addition, 8% of the unmutated smart contracts already contained the vulnerability.

Unchecked low-level call return value and *unchecked send* follow with injection rates of 8.6% and 4.7%, respectively. The frequency of a `call` function is almost double that of a *send* function. It is generally better to use *call* function than `send` but with some important security and implementation considerations. The `call` function is more flexible and allows specifying the amount of gas sent and calls

[3]https://anonymous.4open.science/r/MuSe/
[4]https://figshare.com/s/5473d31f2aead13d2fa8

with data. Nevertheless, it is more vulnerable to reentrancy attacks because it enables the receiving contract to execute arbitrary code.

Finally, *delegatecall to untrusted callee* has the lowest injection rate. The use of the `delegatecall` function is relatively rare but not negligible. It is mainly limited to specific use cases that require advanced behavior, like implementing the Proxy pattern.

By analyzing the injection rate, it is possible to see that some contracts do not have the necessary conditions to inject a given vulnerability. Contracts that lack specific patterns, constructs, or known Solidity functions are intrinsically safe from vulnerabilities that try to exploit these elements. Overall, mutation operators can increase the size of a dataset by 840% by creating new vulnerable versions of smart contracts.

> 🔑 ***RQ$_1$ Summary.*** *The results highlight that the patterns needed to inject "unused return" (71.5%), "authorization via tx.origin" (68%), and "multiple calls in a loop" (56%) are common, whereas those related to "unchecked low-level call return values" (8.6%) and "unchecked send" (4.7%) are less prevalent. The pattern for "delegatecall to untrusted callee" is rare (0.23%) because the delegatecall function is used infrequently and only in specific locations.*

4.2 RQ2. How can the mutants injected by *MuSe* be detected through static analysis?

To answer RQ$_2$, first, we performed an initial detection with Slither on the smartbugs-wild dataset and collected the findings for each contract to have a baseline. Then, we ran Slither on the mutants generated by *MuSe*. Under the assumption that the mutation operator correctly injects the vulnerability, we compared the detection results before and after the mutation. Slither correctly detects a mutant if it is labeled as vulnerable to the type of injected vulnerability. In the case where the contract is already vulnerable and has been mutated, we checked whether, in addition to the pre-existing vulnerability, the injected vulnerability had also been detected by analyzing the lines of code related to the injected vulnerability.

Slither successfully analyzed 335,234 mutants out of the 350,493 generated. Similarly, the execution on the smartbugs-wild dataset failed on 2,700 smart contracts out of 47,398. After excluding the failed executions, we mapped the results achieved by Slither on the original contracts with the mutated ones, if present, and compared the results for further analysis.

Table 3: Detection rate of injected vulnerability and overall performance of Slither on the six considered vulnerabilities.

Vulnerability	TP	FN	Recall	FNR
UC	4,876	0	1.000	0.000
US	3,570	0	1.000	0.000
CL	45,261	10,563	0.810	0.189
UR	124,858	81,184	0.605	0.394
TX	21,765	42,937	0.336	0.663
DTU	15	134	0.100	0.899
-	200,345	134,818	0.597	0.402

As shown in Table 3, the results achieved by Slither against the *unchecked low-level call return value* (UC) and *unchecked send* (US) vulnerabilities are surprisingly high. The tool detected all mutants with the injected vulnerability, showing a recall value equal to 1.00 in both cases. The two vulnerabilities are conceptually similar, which shows that Slither implements strong detectors to check whether the return values of the call and send functions are handled correctly. Slither also performs very well in detecting *multiple calls in a loop* (CL), with a recall of 0.81. While the detection is strong, a noticeable portion of vulnerabilities remains undetected, indicating room for improvement in the detection mechanism of this vulnerability. Performance slows down on *unused return* (UR) with a recall value of 0.63, suggesting that the detection mechanism of these vulnerabilities might be less robust or prone to specific limitations. Performance deteriorates on *authorization via tx.origin* (TX), 0.33 of recall, pointing out significant gaps in detection capabilities for this category. The worst result is detecting *delegatecall to untrusted callee* (DTU) with a recall value of 0.10.

Overall, Slither achieved a recall value of 0.597. The recall value indicates that while Slither effectively identifies certain vulnerabilities, it fails to detect a significant portion (40.2%) of the injected vulnerabilities. The result underscores the need to improve static analysis tools or complement them with additional detection techniques to enhance their accuracy and reduce false negatives.

> 🔑 ***RQ$_2$ Summary.*** *Slither performs very well in some cases but inconsistently across vulnerability types. While it excels in detecting simple vulnerabilities like unchecked low-level call return value and unchecked send, it struggles with more complex vulnerabilities like authorization via tx.origin and delegatecall to untrusted callee. Finally, Slither detected the 59.7% of the injected vulnerabilities using MuSe.*

5 Discussions and Limitations

In this section, we analyze the false negatives resulting from running Slither on the mutated smart contracts. We also discuss the limitations of the study, offering insights about our mutation-based approach to injecting vulnerabilities.

5.1 False Negative Analysis

We analyzed false negatives to extract information about the errors achieved by Slither in detecting vulnerabilities. The way we conducted the experiment allows us to extract only true positives (TP) and false negatives (FN). The mutational operators, validated as described in Section 3, inject the target vulnerability. Having Slither results before and after the mutation, we can analyze whether the tool detects the injected vulnerability (TP) or fails detection (FN).

Authorization via tx.origin (TX). For this vulnerability, the FNR is 0.663. We found some patterns that Slither does not check when detecting this vulnerability type. The most relevant and alarming is related to the Solidity modifier. A Solidity modifier is a reusable function that encapsulates and enforces reusable logic, such as access control or precondition checks, simplifying code and improving maintainability. Figure 2 shows the incorrect implementation of a

modifier used to check the contract owner using `tx.origin` and that Slither cannot label as vulnerable.

Figure 2: Incorrect modifier to restrict owner's access.

```
modifier onlyOwner() {
  require(tx.origin == owner); _;
}
```

Another scenario that Slither fails to detect is when the clause to check the contract ownership is composed of several conditions combined using AND/OR operators. For example, Figure 3 shows the `addPartner` function, which allows the caller to add a new `partner`. The clause in the `require` function ensures that only authorized users (`_dev` or `_owner`) can call this function. The *TX operator* mutated the second occurrence of `msg.sender`, replacing it with `tx.origin`. Using `tx.origin` for authorization introduces a security risk because it refers to the address that initiated the transaction, even if there were intermediate contract calls.

Figure 3: Clause with two conditions combined using OR.

```
function addPartner(address _partner) public {
  require((msg.sender == _dev) || (tx.origin ==
       _owner));
  exchangePartners[_partner] = true;
}
```

Unused return (UR). Slither achieved a false negative rate of 0.394 on this vulnerability. Analysis of false negatives produced an interesting finding. The unused return is correctly detected by Slither when the function being called in the contract is a library function; see Figure 4a. However, the detection fails when the function called is inherited from a contract; see Figure 4b.

Multiple calls in a loop (CL). Slither achieved a false negative rate of 0.189 on this vulnerability. The mutated smart contracts are characterized by several aspects, like function visibility and the presence of modifiers. In addition, the mutation can be injected into the function bodies or in-depth, e.g., nested into an `if` statement. Finally, the statement that undergoes the mutation may be contained in another statement. Nevertheless, we could not find a recurrent pattern in the inconsistent behavior of the detection tool.

Delegatecall to untrusted callee (DTU). The analysis of false negatives revealed two interesting aspects. On the one hand, we noticed that Slither fails to detect simple cases like the one shown in Figure 5. On the other hand, we noticed that most `delegatecall` functions are used in the constructor, which is invoked only at deployment time. Therefore, it is impossible to inject an instance of the vulnerability into the constructor that is exploitable. In addition, the `delegatecall` is often rewritten using the assembly to create a custom function to delegate. Although the mutational operator showed a poor injection rate (0.23%), the impact of this vulnerability is catastrophic if misused in the Proxy pattern. The Proxy pattern uses `delegatecall` to separate state and logic, allowing updates without losing contract data.

Figure 4: Two examples of unused return.

(a) Unused return detected at line 10.

```
library SafeMath {
  function add(uint256 a, uint256 b) internal pure
       returns (uint256) {
    uint256 c = a + b;
    require(c >= a, "addition overflow");
    return c;
  }
}
contract SafeMathExample {
  function addNumbers(uint256 a, uint256 b) public {
    SafeMath.add(a, b);
  }
}
```

(b) Unused return not detected at line 10.

```
contract SafeMath {
  function add(uint256 a, uint256 b) internal pure
       returns (uint256) {
    uint256 c = a + b;
    require(c >= a, "addition overflow");
    return c;
  }
}
contract SafeMathExample is SafeMath{
  function addNumbers(uint256 a, uint256 b) public {
    SafeMath.add(a, b);
  }
}
```

Figure 5: Delegatecall to untrusted callee.

```
address public delegate;
function setDelegate(address _delegate) public {
  delegate = _delegate;
}
function upgradeAndCall(address implementation,
     bytes calldata data) external payable ifAdmin {
  _upgradeTo(implementation);
  (bool success,) = delegate.delegatecall(data);
  require(success);
}
```

5.2 Mutations and Side Effects

This section analyzes the side effects that mutation testing could have on smart contracts. We observed that injecting vulnerabilities through mutational operators often introduces side effects, like code smells. We relied on the Slither official documentation[5] to

[5]https://github.com/crytic/slither/wiki/Detector-Documentation

understand the functionality of its detectors and the labels they use for vulnerabilities.

Unchecked low-level call return value (UC) and Unchecked send (US). Both vulnerabilities are injected using the same approach. The differences are in the conditions used to inject them, but the type of mutation is quite similar. Indeed, the side effects that affect the injection of these vulnerabilities are the same and occur in 75% of the cases. A *deprecated-statement* occurs when outdated constructs are used in a contract, e.g., `throw` instead of `revert`. The two mutation operators, UC and US, remove checks on the return value of a function. When these checks are implemented using an `if` statement, the true branch typically reverts the execution of the function using a `throw` construct. By eliminating the check on the return value, the condition to trigger the `throw` is also removed, resulting in the removal of the `throw` from the mutated code. Since `throw` is deprecated, its removal eliminates using a deprecated statement in 3,772 cases.

Analyzing cases (3,697) where the side effect is the *missing-zero-check* reveals an interesting finding. This issue arises when no validation is performed to ensure that an address, either used as an argument or on which a function is invoked, is not the zero address (*address(0)*). The zero address is often used as a burn address for tokens. When the mutation operator modifies the condition by moving it outside an `if` statement, Slither identifies the presence of a missing zero check. Although the statement remains unchanged, its move outside the `if` condition allows Slither to detect an issue that should be detected regardless of its position in the code.

Authorization via tx.origin (TX). The mutation operator introduces some unexpected behavior in 70% of the cases. When a function performs critical mathematical operations on the contract balance, these operations must be signaled by throwing an event. Functions that perform mathematical operations on the contract balance and do not emit an event are labeled by Slither as `events-math`. When the TX operator mutates them, Slither stops labeling them vulnerable in 2,581 cases. The mutation does not involve the mathematical operations in the contract; therefore, they should continue to be labeled as *events-math*. One possible explanation could be the absence of the msg.sender variable, which seems necessary for a math operation to be signaled by an event.

A similar case involves `events-access`, emitting an event whenever the `msg.sender` variable is used to change the owner of the contract. In 2,398 cases, it is understandable to eliminate this vulnerability when `msg.sender` is replaced with `tx.origin`.

The last side effect, called by Slither *suicidal*, occurs 1,058 times and concerns using the `selfdestruct` function, which should be restricted to the contract owner. When the mutation replaces `msg.sender` with `tx.origin` in the modifier used to give access to `selfdestruct`, Slither correctly detects the misuse of `selfdestruct`. The detector applies the appropriate control, discussed in the False Negative Analysis section—specifically, verifying the contract owner using a modifier. However, the detection behavior appears inconsistent depending on the functionality implemented within the function. For instance, if the function includes the invocation of a well-known operation, such as `selfdestruct`, Slither verifies access to the function by analyzing the associated modifier. Conversely, Slither fails to check the modifier if the function does not include recognized features. This inconsistency highlights a significant issue: regardless of the functionality implemented, an incorrect modifier implementation should always be detected, which is especially critical when the modifier is responsible for ensuring that the function invoker is the contract owner.

Unused return (UR). We noted that the mutation operator introduces other vulnerabilities or code smells alongside the injected vulnerabilities in 67% of the cases. For example, in some cases, the operator splits a single statement containing declaration and initialization into two separate statements, one for declaring the variable and one for initializing it. The initialization is then made vulnerable if the return value of a function is used to initialize the variable. The return value is not assigned to the variable, thus creating an unused return and leaving the variable uninitialized. The side effect of this mutation is the creation or removal of some code smells. The mutation has introduced *uninitialized-state* (27,890), *initialized-local* (97,932), or *constable-states* (28,352). While not expected, these side effects are understandable, given the type of mutation introduced. Variables uninitialized due to the mutation introduce *uninitialized-state* and *uninitialized-local*, depending on the scope of the uninitialized variable. In cases where the variable affected by the mutation undergoes no other changes in the code, we have a *constable-states*, a variable not declared constant.

Interesting side effects are those that remove some vulnerabilities or code smells present before the mutation and that were removed by the mutation. For example, *divide-before-multiply* (-3,913) is resolved when the return value of multiplication is not assigned to the variable in which the result of the previous division was saved. Another is *incorrect-equality* (-3,851), which occurs when a variable to which the contract balance is assigned is used in a strict equality. Removing the contract balance assignment from the variable also removes the vulnerability. Then we have *reentrancy-benign* (-5,340), *reentrancy-no-eth* (-3,191), *reentrancy-eth* (-830), and *reentrancy-unlimited-gas* (-818), all of which are removed from the mutation because by not assigning the return value to a variable involved in reentrancy, it does not change state and does not create the preconditions for reentrancy. Additionally, *controlled-array-length* was added and removed depending on the case. It was introduced in 371 mutants but removed in 790. It was eliminated when the array index was not initialized using the return value of a function and introduced when the index remained uninitialized.

Interestingly, some vulnerabilities or code smells emerged, although not directly connected to the mutation. For example, *external-function* or *timestamp* were introduced by the UR operator but appeared in different lines of code than those affected by the mutation. The mutated statement and the statement affected by the newly emerged vulnerability shared no common functions or variables. We hypothesize that these instances represent false positives, potentially caused by conflicts within the Slither execution flow or interactions between its detectors.

Multiple calls in a loop (CL). The CL operator introduces side effects that are strictly related to the vulnerability injected in 30% of the cases. It introduces *msg-value-loop* (6617), *costly-loop* (451), and *cyclomatic-complexity* (126). The first case occurs when `msg.value` is used in mathematical operations inside a loop. The second occurs when we use costly operations inside a loop that might waste gas, so

optimizations are justified. The last case occurs when the cyclomatic complexity is higher than 10. In this case, all side effects are closely related to the mutation and, in some cases, are unavoidable.

Delegatecall to untrusted callee. The DTU operator introduces *missing-zero-check* and *naming-convention*. The mutation introduces a new address called *delegate* and a new function to set the address called *setDelegate*. The setter function does not check that the address is not the zero address (*address(0)*). The *naming-convention* is a warning that refers to the address argument of *setDelegate*; it is not in *mixedCase* violating the Style Guide of Solidity[6].

> **Discussion Summary.** *Slither exhibits inconsistent behavior and misses vulnerabilities in specific scenarios, suggesting potential gaps in its detection algorithms. Furthermore, its ability to detect some warnings depends on the presence of specific variables, even when these variables are not directly related to the core logic of the vulnerability.*

6 Threats To Validity

This section describes the potential threats to validity, including construct, internal, external, and conclusion validity.

Internal Validity. The vulnerability injection approach threatens the validity of our study. The initial version of our mutation operators exhibits side effects, which can impact the results. As discussed in Section 5, some side effects are understandable and legitimate, others are unavoidable, and some can be mitigated by refining the mutation operators.

Another threat concerns the relationship between the injected vulnerability and its side effects. Sometimes, the code smells that emerged as side effects were not attributable to the injected vulnerability. Mutations could introduce code smells independently of the injected vulnerability. We have discussed all observed side effects and explained their underlying causes, aiming to increase awareness about their proper usage and highlight areas for improvement in the mutation operators.

External Validity. We relied on Slither [16] as the static analysis tool to detect injected vulnerabilities, which may limit the generalizability of our findings regarding static analysis tools. However, Slither is one of the most widely used tools in smart contract analysis, recognized for its fast execution time and popularity in academic and industrial settings. It is open-source, actively maintained, and readily accessible, making it ideal for this study. Importantly, the experiment could be easily extended to evaluate additional tools.

Another external threat to validity is the focus on only six vulnerabilities. However, these vulnerabilities were selected because they are among the most frequently discussed in the literature [38] and represent common security issues in smart contracts. In addition, the tool can be easily extended by implementing new mutation operators to inject new vulnerabilities.

Construct Validity. A threat to construct validity concerns the analysis of false negatives. An exhaustive analysis of all false negatives would take too much time and might reveal cases we did not consider. We restricted the false negative analysis to observing a statistically significant sample from which we extracted the findings reported in the paper.

A second threat concerns using SmartBugs [12], which accelerates the experiments and simplifies the analysis of the results. We followed its official documentation, which lists some issues and discusses how to mitigate them. Nevertheless, we acknowledge that our study could have been threatened by relying on this framework.

Another threat to construct validity relates to *MuSe* implementation. Our tool is based upon SuMo [2], a widely recognized and tested mutation testing tool that has already been extended for regression mutation testing [4]. However, by relying on SuMo, *MuSe* may inherit its defects, potentially impacting the performance of *MuSe*. To mitigate the potential issue, we manually validated our extensions using a statistically significant set of smart contracts composed of a random subset of the smartbugs-wild dataset.

Lastly, choosing the dataset for our experiments and validation introduces a potential threat to external and construct validity. We selected the smartbugs-wild dataset [15], one of the most widely used datasets in the literature. The dataset is recognized for its considerable size and frequent use in empirical investigations, reinforcing its relevance and reliability for our study.

7 Conclusion

This paper explores a mutation-based approach to inject known vulnerabilities into smart contracts to generate new benchmarks to evaluate vulnerability detection tools. We proposed *MuSe*, a mutation seeding tool that implements mutational operators that identify the appropriate pattern in which to inject the mutation. *MuSe* is capable of injecting six vulnerabilities. An injection rate characterizes each vulnerability since not all smart contracts have the conditions to be mutated and thus be vulnerable to a problem. *MuSe* has been validated and is easily extended by adding new mutational operators. Slither, a static analyzer, analyzed smart contracts generated by mutation. The results showed gaps in Slither detectors, showing the current limitations of static analyzers and leaving room for improvement. Our study highlights that using new benchmarks generated through a mutation-based approach can improve the validation of static analysis tools.

As part of our future work, we aim to enhance *MuSe* by introducing additional mutation operators and refining existing ones to reduce unintended behavior. Our ultimate goal is to develop a fully automated tool capable of generating vulnerable smart contracts starting from an initial set of smart contracts. We plan to address the gaps identified by Bobadilla *et al.* [6] by creating new datasets. The current literature offers limited labeled benchmarks, most of which consist of smart contracts written in older versions of Solidity. Using our mutation seeding tool, we intend to inject vulnerabilities into audited and updated smart contracts to produce new, labeled benchmarks and fill these critical gaps.

Acknowledgments

Finanziato dall'Unione Europea - Next Generation EU, Missione 4 Componente 1 CUP D53D23008400006.

[6] https://docs.soliditylang.org/en/latest/

References

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. 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, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*. Springer, 164–186.

[2] Morena Barboni, Andrea Morichetta, and Andrea Polini. 2021. SuMo: A Mutation Testing Strategy for Solidity Smart Contracts. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. 50–59. doi:10.1109/AST52587.2021.00014

[3] Morena Barboni, Andrea Morichetta, and Andrea Polini. 2022. SuMo: A mutation testing approach and tool for the Ethereum blockchain. *Journal of Systems and Software* 193 (2022), 111445. doi:10.1016/j.jss.2022.111445

[4] Morena Barboni, Andrea Morichetta, Andrea Polini, and Francesco Casoni. 2024. ReSuMo: a regression strategy and tool for mutation testing of solidity smart contracts. *Software Quality Journal* 32, 1 (2024), 225–253.

[5] Kent Beck. 2022. *Test driven development: By example*. Addison-Wesley Professional.

[6] Sofia Bobadilla, Monica Jin, and Martin Monperrus. 2025. Do Automated Fixes Truly Mitigate Smart Contract Exploits? *arXiv preprint arXiv:2501.04600* (2025).

[7] Vitalik Buterin et al. 2014. Ethereum white paper: a next generation smart contract & decentralized application platform. *First version* 53 (2014).

[8] Patrick Chapman, Dianxiang Xu, Lin Deng, and Yin Xiong. 2019. Deviant: A Mutation Testing Tool for Solidity Smart Contracts. In *2019 IEEE International Conference on Blockchain (Blockchain)*. 319–324. doi:10.1109/Blockchain.2019.00050

[9] Ting Chen, Yuxiao Zhu, Zihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhange. 2018. Understanding Ethereum via Graph Analysis. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 1484–1492. doi:10.1109/INFOCOM.2018.8486401

[10] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, and Shunhui Ji. 2024. SGDL: Smart contract vulnerability generation via deep learning. *Journal of Software: Evolution and Process* 36, 12 (2024), e2712.

[11] Etienne Daspe, Mathis Durand, Julien Hatin, and Salma Bradai. 2024. Benchmarking Large Language Models for Ethereum Smart Contract Development. 1–4. doi:10.1109/BRAINS63024.2024.10732686

[12] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. 2023. SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2102–2105. doi:10.1109/ASE56229.2023.00060

[13] Monika Di Angelo and Gernot Salzer. 2019. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON)*. IEEE, 69–78.

[14] Ardit Dika and Mariusz Nowostawski. 2018. Security Vulnerabilities in Ethereum Smart Contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 955–962. doi:10.1109/Cybermatics_2018.2018.00182

[15] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

[16] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[17] Asem Ghaleb and Karthik Pattabiraman. 2020. ‘. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 415–427. doi:10.1145/3395363.3397385

[18] Ákos Hajdu, Naghmeh Ivaki, Imre Kocsis, Attila Klenik, László Gönczy, Nuno Laranjeiro, Henrique Madeira, and András Pataricza. 2020. Using Fault Injection to Assess Blockchain Systems in Presence of Faulty Smart Contracts. *IEEE Access* 8 (2020), 190760–190783. doi:10.1109/ACCESS.2020.3032239

[19] Gerardo Iuliano and Dario Di Nucci. 2024. Smart Contract Vulnerabilities, Tools, and Benchmarks: An Updated Systematic Literature Review. arXiv:2412.01719 [cs.SE] https://arxiv.org/abs/2412.01719

[20] Y. Ivanova and A. Khritankov. 2020. RegularMutator: A Mutation Testing Tool for Solidity Smart Contracts. *Procedia Computer Science* 178 (2020), 75–83. doi:10.1016/j.procs.2020.11.009 9th International Young Scientists Conference in Computational Science, YSC2020, 05-12 September 2020.

[21] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.* 37, 5 (Sept. 2011), 649–678. doi:10.1109/TSE.2010.62

[22] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 259–269.

[23] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. *IEEE Access* 10 (2022), 6605–6621. doi:10.1109/ACCESS.2021.3140091

[24] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access* 10 (2022), 6605–6621.

[25] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 65–68.

[26] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[27] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. 2019. Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. *Journal of Cases on Information Technology (JCIT)* 21, 1 (2019), 19–32.

[28] Alexander Mense and Markus Flatscher. 2018. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th international conference on information integration and web-based applications & services*. 375–380.

[29] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. Advances in Computers, Vol. 112. Elsevier, 275–378. doi:10.1016/bs.adcom.2018.03.015

[30] Reza M Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. *arXiv preprint arXiv:1809.02702* (2018).

[31] Heidelinde Rameder, Monika Di Angelo, and Gernot Salzer. 2022. Review of automated vulnerability analysis of smart contracts on Ethereum. *Frontiers in Blockchain* 5 (2022), 814977.

[32] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. 2021. Empirical evaluation of smart contract testing: What is the best choice?. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 566–579.

[33] Noama Fatima Samreen and Manar H Alalfi. 2021. Smartscan: an approach to detect denial of service vulnerability in ethereum smart contracts. In *2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 17–26.

[34] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart contract: Attacks and protections. *Ieee Access* 8 (2020), 24416–24427.

[35] Fabian Schär. 2021. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review* (2021).

[36] Fernando Richter Vidal, Naghmeh Ivaki, and Nuno Laranjeiro. 2024. OpenSCV: an open hierarchical taxonomy for smart contract vulnerabilities. *Empirical Software Engineering* 29, 4 (2024), 101.

[37] Xiaoyin Wang, Jiaze Sun, Chunyang Hu, Panpan Yu, Bin Zhang, and Donghai Hou. 2022. EtherFuzz: mutation fuzzing smart contracts for TOD vulnerability detection. *Wireless Communications and Mobile Computing* 2022, 1 (2022), 1565007.

[38] Oualid Zaazaa and Hanan El Bakkali. 2023. A systematic literature review of undiscovered vulnerabilities and tools in smart contract technology. *Journal of Intelligent Systems* 32 (09 2023). doi:10.1515/jisys-2023-0038

[39] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. 2024. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. *IEEE Trans. Softw. Eng.* 50, 6 (June 2024), 1360–1373. doi:10.1109/TSE.2024.3383422

[40] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2021. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2084–2106. doi:10.1109/TSE.2019.2942301