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

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.

1. Introduction

Starting from their first appearance the adoption of blockchain technologies has drastically increased, and novel application scenarios are constantly investigated. This has greatly to do with the inherent trust that can be brought, in some contexts, by the adoption of these technologies. Moreover, in recent years many blockchain technologies have been augmented with support for smart contracts. These can be considered as software programs that can be deployed and executed over a blockchain and that, during their execution, will generate data (transactions) that will be stored within the blockchain itself. Such smart contracts permit the stakeholders that take part in the smart contract execution, to have enough guarantees so as to trust that the activities performed by the other interacting parties are in line with what is specified in the contract itself. Such mechanisms have been fruitfully adopted in the engineering of software systems (Porru et al., 2017), and in particular in the integration of inter-organizational and collaborative software systems (Corradini et al., 2020, 2022). Compared to traditional software systems, the development of smart contracts presents unique characteristics and challenges as a consequence of the underlying deployment and execution environment (Zheng et al., 2017; Destefanis et al., 2018). There is then a need for novel and appropriate testing tools that allow the developer community to write and deploy safer code. In particular, it is possible to identify several reasons why smart contracts generally ask for high-reliability guarantees, and a thorough testing process. The following is a, probably non-exhaustive, list of relevant characteristics (Barboni et al., 2021):

- **Smart Contracts manage valuable assets**: Smart contracts can control large amounts of cryptocurrency and other valuable assets. Deploying faulty code can result in the accidental loss of the assets held by the contract. The potential financial gain, and the anonymous nature of the blockchain further act as an incentive for attackers. Even a small loophole in the code can allow malicious users to drain large amounts of funds. A typical example is the famous DAO attack (Mehar et al., 2019), in which a reentrancy vulnerability

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: morena.barboni@unicam.it (M. Barboni), andrea.morichetta@unicam.it (A. Morichetta), andrea.polini@unicam.it (A. Polini).

https://doi.org/10.1016/j.jss.2022.111445
0164-1212/

was exploited to transfer 3.6 million Ether (around $50 million).

- **Transactions are irreversible**: Smart contracts are deployed and executed in the blockchain environment, which does not allow to revert transactions. A transaction becomes irreversible once it receives enough confirmations from the network. At the same time, it is not possible to recover assets lost during the smart contract execution.
- **Smart Contracts are immutable**: Smart contracts feature an anomalous development life cycle (Miraz and Ali, 2020) that cannot be represented by traditional software development models. Enhancing the code or fixing bugs after deployment is not possible. Immutability ensures that the code is tamper-proof, but it also prevents further upgrading and code maintenance. Correcting a bug after deployment is a very costly operation since it requires the creation of a brand new contract on the blockchain.
- **Blockchain environment**: The smart contract execution is dependent on the underlying blockchain platform and the possible interactions with other cooperating contracts. Developers must carefully consider these relationships and the peculiarities of the new distributed environment to write safer code.
- **New software stack**: The smart contract's execution environments and programming languages (e.g. Solidity) are relatively young and continuously evolving; many issues and vulnerabilities are still being discovered. Such characteristics make it harder for developers to program with confidence and to write safe code.
- **Lack of best practices**: Zou et al.'s (Zou et al., 2021) recent interview with smart contract experts exposed a lack of best practices for writing reliable contract code. Finding code examples and development standards is particularly difficult when working on new applications. This issue can cause developers to pick up bad programming habits and dangerous anti-patterns. An anti-pattern is a solution created in response to a recurring problem, which appears to be appropriate and effective. However, it ends up being ineffective or even counterproductive.
- **Lack of mature testing tools**: Smart contract development cannot count on the wide selection of testing tools available for traditional software. The currently available tools are not as mature, and in some cases, they are ineffective at ensuring the quality of the contract code. Zou et al.'s (Zou et al., 2021) research shows that the developer community is especially interested in code auditing tools, which help in discovering bugs and vulnerabilities.

The testing activity allows deriving confidence in the correctness and reliability of a smart contract. However, this level of confidence is strictly dependent on the adequacy of the implemented test suite. Most white-box adequacy criteria are defined on the basis of code coverage, which measures how thoroughly the logical elements within the software are exercised by the test suite. If certain elements are not covered by tests, they might contain faults that persist in the system after testing. Even though code coverage helps in identifying under-tested parts of a program, research shows that it does not represent a good indicator of the test suite effectiveness (Inozemtseva and Holmes, 2014; Tengeri et al., 2016), and that mutation testing can be considered the strongest test criterion to characterize high-quality test suites, even though it leads to higher costs (Frankl et al., 1997). Nonetheless, the specific characteristics of smart contracts listed above, make clear that by their very nature smart contracts can be generally considered business-critical artifacts, both since they often directly relate to financial assets, and as a consequence of the cost associated with each transaction generated by a contract in relation to its mining, as will be detailed in the next section. In such a context the inherent costs of testing activities are generally accepted and compensated by the reduced business risks that a thoughtful testing activity can bring.

This paper proposes a novel mutation testing approach tailored for smart contracts to be deployed on the Ethereum blockchain. The approach has been implemented in SuMo (i.e., a mutated version of SoMu, which stands for **So**lidity **Mu**tator) and applied to assess the test suites associated to open source and publicly available smart contracts. The work presented here extends the work reported in (Barboni et al., 2021) in many different directions. In particular, we provide here a more detailed discussion and analysis of the mutation operators, that have been now implemented in a freely available service for which general architectural aspects are presented. Finally, we report here a wide validation of the approach, and we compare its effectiveness with respect to alternative proposals.

The rest of the paper is organized as follows. Section 2 includes background material about the Ethereum blockchain and the mutation testing technique, while Section 3 presents the research questions that we aim to answer in this work. In Section 4 we describe the SuMoapproach and we provide details about the included mutation operators. In Section 5 we present the SuMo architecture and implementation while in Section 6 we illustrate the validation activity we performed. Section 7 describes potential threats to the validity of the study, while Section 8 reports those works that are closer to our proposal. Finally, in Section 9 we give a brief summary of the findings and identify opportunities for further research.

2. Background

In this section we provide some background knowledge about Ethereum blockchain and smart contracts (Section 2.1). We also report information about the mutation testing technique (Section 2.2) and its application. Mutation testing is a very broad topic with many open challenges (Papadakis et al., 2019) and research directions. There are several interesting aspects to be analyzed, such as test data generation (Dang et al., 2021, 2020; Yao et al., 2020) and the impact of test flakiness on the reliability of the approach (Shi et al., 2019). However, an in-depth discussion of all the connected aspects and techniques would require an extensive analysis of the literature. Therefore, Section 2.2 mostly provides general knowledge about the mutation testing process and the connected cost reduction techniques.

2.1. Ethereum smart contracts

A smart contract is a program that runs on top of a blockchain network. Smart contracts are commonly referred to as 'self-enforcing agreements' because the execution process automatically enforces the terms defined within the code. This property allows the execution of transactions among untrusted parties without the need for a central authority. Each contract can implement complex business logic to carry out transactions that go beyond a simple monetary transfer. **Ethereum** is a blockchain technology specifically built for the creation and execution of smart contracts, and it includes supporting mechanisms for Solidity, a Turing-complete programming language.

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

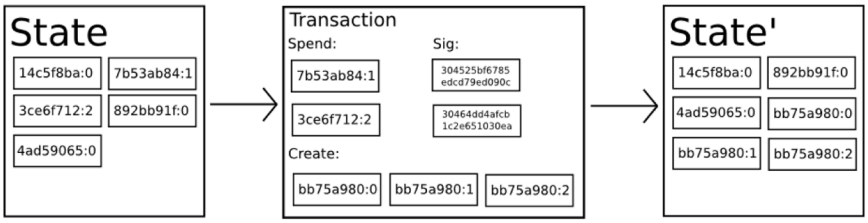

Fig. 1. Ethereum state transition function.

- **Externally Owned Account (EOA)**: users of Ethereum have to create accounts to execute transactions. For each EOA, a unique public–private key pair is generated; the private key grants the user control over the account funds, while the public key becomes the identity of the account itself. Each EOA can send transactions both to another EOA or to a Contract Account.
- **Contract Account (CA)**: a smart contract is not accessible until it gets deployed on the Ethereum network, creating a Contract Account. Other accounts can then interact with the CA through its associated address. A contract can perform operations on its storage and fire calls to other contracts, but only in response to a transaction.

Every account on Ethereum is characterized by an **Account State** which maintains the following information:

- **nonce**: The nonce stores either the number of transactions executed from an EOA or the number of generated contract instances for a CA;
- **balance**: The balance represents the amount of Ether owned by the account;
- **storageRoot**: The storageRoot attribute is only relevant for a CA. It contains the Merkle-Patricia trie root that encodes the storage contents of the account;
- **codeHash**: the codeHash holds the hash of the Solidity code associated to the specific CA.

Ethereum world state. Ethereum was defined in the Ethereum Yellow Paper (Wood, 2021) as a **transaction-based state machine**. Unlike Bitcoin, Ethereum has a global state that enables the management of real account balances. Each Account State is stored inside a **World State trie** data structure. Every time transactions, contract executions, and mining activities take place, the global state of the blockchain is updated. The Ethereum blockchain can be seen as a state transition system, where a state transition takes a state and a transaction and outputs a new state as a result as exemplified in Fig. 1. The last block of the chain represents the current world state. This block holds information about the balances of all the Ethereum accounts after the most recent transactions have been executed.

Ethereum transactions. When a transaction is executed the state of the blockchain is permanently modified. There are three types of transactions that can be executed with different purposes:

- **Monetary**: A monetary transaction is a simple transfer of funds between EOAs.
- **Contract Deployment**: A contract deployment transaction allows the sender to create a Contract Account on the blockchain. The transaction must contain the compiled contract code as a payload. The newly created CA will be accessible through an address derived from the creator's address and nonce.
- **Contract Execution**: A contract execution transaction runs a function defined within a deployed contract. Contract execution is always initiated by an EOA. Upon execution, the smart contract can read or write to its own state and send message calls to other CAs.

The execution and confirmation of a transaction require several steps. First, the sender must create a transaction and broadcast it to the Ethereum network. The miner nodes are responsible for including new transactions into a **candidate block**. Each miner selects the transactions to include in the block from a **transaction pool** and starts validating them. This process involves checking the digital signature and the output of any contract code execution. Once the candidate block is built, the miner starts working to find a valid Proof of Work solution. When a valid hash is found the candidate block is broadcast to the rest of the network. Upon reception of the block, each full node repeats the validation process for each transaction. The nodes do not compare the results among themselves, but they rely solely on the **consensus mechanism**; if at least one transaction is invalid every honest node will reject the block. A valid block cannot be considered part of the main chain until it is confirmed. The distributed nature of the blockchain can cause the nodes to receive blocks in a different order. When separate parts of the network follow distinct solutions the main chain **forks**. Ethereum selects the only valid branch using the **GHOST (Greedy Heaviest Observed Subtree)** protocol (Buterin, 2013). If a valid block is part of a discarded fork, all the included transactions go back to the transaction pool. Otherwise, the transactions will be permanently committed to the blockchain. In this case, users must wait for a confirmation which is generally issued after 12 blocks.

Gas mechanism. Each node of the Ethereum network includes the **Ethereum Virtual Machine (EVM)**, a stack-based execution environment for running smart contract code. The EVM allows each Ethereum node to participate in the verification protocol. Since each contract is locally executed by the whole network the validation process can become computationally expensive. Moreover, non-terminating program executions could potentially freeze the participating nodes. Since the EVM supports Turing complete languages, predicting the amount of required computational resources is not simple. Ethereum implements a **gas mechanism** that serves two main purposes: It limits the usage of resources and compensates miners for their work. The issuer of a transaction must pay a gas fee to the client that commits the transaction to the blockchain. This fee is determined by the amount of computation and data storage required by the transaction execution. Each transaction object contains a `gasLimit` and a `gasPrice` field. The **gas price** indicates the market price in Wei of a unit of gas. The **gas limit** is the maximum amount of gas that can be burnt for executing the transaction. The amount of gas needed for executing a smart contract depends on the number and type of instructions run by the EVM. When a smart contract is written in a high-level programming language (e.g. Solidity), the code has a certain level of complexity. The smart contract code is then compiled into **bytecode**, a low-level language that can be executed by the EVM. The bytecode can be disassembled into **opcodes** (operation codes), which are human-readable, low-level instructions. Each opcode[1] is associated with a cost expressed

[1] https://ethervm.io/

in terms of gas. More complex contracts will require more gas units for executing the code. Charging fees per computational step encourages developers to optimize the code and avoid the inclusion of costly patterns.

Solidity. Solidity is the most widely used object-oriented language for developing smart contracts. When the high-level Solidity code is compiled to bytecode, it can be run on the EVM to enforce the terms described within the contract. The syntax of Solidity is mainly influenced by JavaScript, however, it is statically and strongly typed. Solidity supports inheritance, polymorphism, and user-defined data types. Moreover, it introduces several novel constructs and keywords dedicated to smart contracts development.

2.2. Mutation testing

Mutation testing is a white-box, fault-based testing technique for evaluating and enhancing the quality of a test suite. Mutation testing aims to guarantee the adequacy of test data in finding real faults by purposefully introducing small defects into copies of the source code. This process helps to identify limitations in the implemented test suite and to ensure the deployment of more reliable code. Mutation testing is based on the assumption that a subset of simple faults is sufficient to simulate the majority of faults that can be found in a real-world program.

The mutation testing process. The traditional mutation process generates a set of faulty copies of the original program P called **mutants**. A mutant does not deviate largely from the original program, but it contains a minor alteration that simulates a typical programming mistake. The core element of mutation testing is a set of rules called **mutation operators**. Each operator specifies how the source code must be altered to create a mutant. For instance, let us consider the Solidity function in Listing 1. The owner of a smart contract can invoke this method to transfer contract ownership to another address (i.e., `newOwner`). The `onlyOwner` modifier[2] attached to the function prevents unauthorized users from performing this operation; if the developer forgot to add it, anyone could accidentally (or maliciously) transfer ownership to another address. Here, a mutation operator can emulate the developers' mistake by depriving the function of its modifier.

Listing 1: Example of mutant for the Solidity language

```
1  //Original
2  function transferOwnership(address newOwner) public
       onlyOwner {
3      _transferOwnership(newOwner);
4  }
5
6  //Mutant
7  function transferOwnership(address newOwner) public  {
8      _transferOwnership(newOwner);
9  }
```

The adequacy assessment is performed by running the test suite T against each mutant program, and verifying whether the tests can detect the fault injected into the code. If at least one test case in T is able to detect the fault, the mutant is said to be **killed**, otherwise, it is marked as **live**. Some mutants can also be trivially killed by the compiler. If the mutation injected into the program causes a compilation error, the mutant is considered **stillborn**. Listing 2 shows an example of such mutants for the Solidity language. Here, a mutation operator replaced the `public` visibility keyword of the `deposit` function with each variant (`external`, `internal` and `private`), generating 3 mutants. In the general case, each one of these mutants can be successfully compiled. However, since the `deposit` function is declared as `payable`, it must be able to receive Ether and thus it must have either `public` or `external` visibility[2]. In this scenario, only the first mutant is valid, whereas mutants two and three will be marked as stillborn. These invalid mutants represent an overhead for the mutation testing process, as they waste computational resources without providing useful insights about the quality of T.

Listing 2: Example of function visibility mutations

```
1  //Original
2  function deposit(uint256 amount) payable public {
3      require(msg.value == amount);
4  }
5
6  //Mutant 1 - Valid
7  function deposit(uint256 amount) payable external {
8      require(msg.value == amount);
9  }
10
11 //Mutant 2 - Stillborn
12 function deposit(uint256 amount) payable internal {
13     require(msg.value == amount);
14 }
15
16 //Mutant 3 - Stillborn
17 function deposit(uint256 amount) payable private {
18     require(msg.value == amount);
19 }
```

Additionally, some mutants might generate **infinite loops**. If the execution of the test suite does not terminate, the mutant is considered **timed out**. These mutants must be ignored as they do not provide information about the quality of T.

The percentage of mutants killed by T (Eq. (1)), called **Mutation Score (MS)**, represents the adequacy value of the test suite. The developer can iteratively improve the test suite until the adequacy criterion is satisfied by the Mutation Score.

$$MS = \frac{valid\ mutants - surviving\ mutants}{valid\ mutants} \times 100 \tag{1}$$

However, before calculating the Mutation Score, it is necessary to identify and remove the equivalent mutants. An **equivalent mutant** is a program that contains syntactic modifications, but it behaves exactly like the original program. Listing 3 shows an equivalent mutant generated from a real-world smart contract; here, a mutation operator that targets binary expressions changed the mathematical operator $*$ (line 4) into / (line 11). Even so, the value of the `MIN_DURATION` constant is not affected. Undetected equivalent mutants decrease the reliability of the Mutation Score because they are mislabeled as faulty programs. As we summarize in Table 1, the union of **stillborn**, **timed-out** and **equivalent** mutants constitute the set of **invalid** mutants that should not be considered in the calculation of the Mutation Score (MS).

Listing 3: Example of equivalent mutant

```
1  //Original
2  contract CrowdfundingCampaign {
3      ...
4      uint public constant MIN_DURATION  = 60 * 60 * 1;
5      ...
6  }
7
8  //Mutant
9  contract CrowdfundingCampaign {
10     ...
11     uint public constant MIN_DURATION  = 60 * 60 / 1;
12     ...
13 }
```

[2] Solidity documentation: https://docs.soliditylang.org/.

The analysis of the **valid live** mutants can provide useful insights into the types of faults that cannot be caught by the existing tests. Based on this feedback it is possible to enhance the test suite and cover scenarios that were neglected during the test design phase. However, the set of valid mutants might contain **subsumed** (or redundant) mutants. A subsumed mutant is defined on the basis of a subsumption relationship (Kurtz et al., 2014): a mutant *m1* subsumes another mutant *m2*, if whenever m1 is detected by a test, m2 is also detected. Thus, a subsumed mutant entails the execution of a (possibly complete) test run without providing new information to the tester, skewing the Mutation Score.

Listing 4 shows an example of a possibly subsumed mutant for Solidity smart contracts. Here, the `decommission()` function was implemented to permit the deletion of the Coinosis contract from the blockchain. To prevent unauthorized deletions, the `require` statement on line 7 checks whether the function was invoked by the contract owner. If not, the transaction is reverted with an error message. To ensure the safety of `decommission()`, a tester must verify that it cannot be executed from a non-owner account. Here, a mutation operator simulates two faults. The first mutation rule comments out the `require` statement to prevent the authorization check from taking place. The same operator also replaces `require` with an `assert` statement, another error handling statement that should only be used for internal error checking[2] to possibly identify bugs. Unlike `require`, the `assert` statement does not support error messages, does not refund unused gas to the sender, and it leads to an run-time failure. Note that, considering the exception handling policies foreseen by the Solidity language, both rules will generate a mutant that can be successfully compiled. In this scenario, any test capable of killing *Mutant1* will kill *Mutant2* as well. In fact, *Mutant1* will prevent the check on the address of the caller from taking place altogether. In such a case, *Mutant2* is said to be subsumed by *Mutant1*.

Listing 4: Example of subsumed mutant

```
1  //Original
2  contract Coinosis {
3
4    string constant NOT_OWNER = "The sender account is not the
         owner";
5
6    function decommission() public {
7      require(msg.sender == owner, NOT_OWNER);
8      selfdestruct(owner);
9    }
10
11   //Mutant1
12   function decommission() public {
13     //(msg.sender == owner, NOT_OWNER);
14     selfdestruct(owner);
15   }
16
17   //Mutant2
18   function decommission() public {
19     assert(msg.sender == owner);
20     selfdestruct(owner);
21   }
22
23 }
```

Cost reduction techniques. Mutation analysis is known for being a computationally expensive, albeit powerful, testing technique. A full-scale mutation campaign can be extremely costly and time consuming, especially in the case of large code bases with equally large test suites. Researchers and practitioners have been investigating appropriate cost reduction techniques for a long time (Jia and Harman, 2011; Zhang et al., 2010). In order to make the mutation testing process less cumbersome, it is generally possible to act either on the mutant generation, or on the cost of the program execution (Jia and Harman, 2011). Both classes of techniques must be applied carefully because they pose a threat to the effectiveness of the final test suite.

Table 1
Types of mutants that can be generated during mutation testing.

Mutant Category	Mutant Type	Description
Valid	Live	Mutant for which all tests pass
	Killed	Mutant for which at least one test fails
	Subsumed (or Redundant)	Mutant whose detection can be predicted on the r basis of the detection of other mutant(s)
Invalid	Equivalent	Mutant that behaves like the original program
	Stillborn	Mutant that causes a compilation error
	Timed Out	Mutant that prevents the test suite from terminating

Most cost reduction techniques act on the number of mutants to be tested, such as **mutant sampling**, **mutant clustering** or **selective mutation**. While the former pick a subset of mutants to be tested, selective mutation omits those operators that are empirically found to be the least effective, such as the ones that generate a large amount of stillborn or redundant mutants. If done properly, the selection of a few operators can result almost as effective as the usage of the whole initial set. Unlike the previous techniques, **higher-order mutation testing** reduces the number of mutants by merging multiple mutations together. Although applying multiple changes to the same program results in a smaller set of mutants, it can also cause a significant loss of test sensitivity (Kintis et al., 2010). Lastly, the **Trivial Compiler Equivalence** (TCE) (Papadakis et al., 2015) technique leverages the compiler technology to detect equivalent and redundant mutants. The TCE marks any two programs with identical binaries as equivalent. Besides reducing the total number of test runs, removing such mutants helps to obtain a more reliable Mutation Score.

On the other hand, the execution cost of a mutant program can be reduced by loosening the conditions that must be satisfied to kill it. In **weak mutation testing**, a mutant is considered killed if the injected mutation caused it to reach a different state than the original program. Since weak mutation testing does not require running the entire program, it can significantly reduce the test execution time. However, the test adequacy assessment will not be as precise.

The other main problem of mutation testing lies in the amount of **manual effort** required to the tester. The manual analysis of each live mutant is an essential precondition for the improvement of the test suite. However, there is an additional cost tied to the detection of the equivalent mutants, which is an undecidable problem. Techniques like higher-order mutation testing, selective mutation, and the Trivial Compiler Equivalence (Papadakis et al., 2015) can help to alleviate this issue. Nevertheless, the filtering process still requires human intervention, as no automated technique is capable of removing the entirety of equivalent mutants.

3. Research questions

In the following, we present the research questions that we aim to answer in this work.

RQ1: Do the SuMo-specific and solidity-specific operators contribute to the identification of weaknesses in smart contract test suites? Our main objective is to assess whether the novel and the Solidity-specific operators implemented by SuMo contribute to the improvement of the test suite. To answer this question, we will analyze the Mutation Score achieved by the test suite with respect to the relevant mutation operators. The general assumption is that small values for the Mutation Score are a symptom of missing or poorly designed test cases. Nevertheless, the generated mutants could also mimic bugs that are not that dangerous or likely to occur, so that the effort spent is not worth it. Thus, we analyze and discuss the live mutants generated by each operator, paying particular attention to those that are rarely killed throughout the experiment.

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

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

4. SuMo mutation approach

One of the main challenges of implementing a mutation testing approach is designing a comprehensive set of **mutation operators** that can simulate a wide range of faults. In fact, a test suite can be improved to detect those mutants that remained undetected; an approach based on a limited fault model cannot ensure the derivation of a high-quality test suite. The second challenge concerns **usability** and **effectiveness**. As it is well known, mutation testing is a rather expensive activity. Even though the specific application context can somehow accept higher verification costs, in the definition of the SuMo approach we tried to introduce mechanisms that permit to reduce the overall effort needed to assess a test suite, without sacrificing much of its effectiveness. In the following we present the main points of the SuMo mutation approach.

Fault model. To build the fault model of SuMo we carried out an in-depth analysis of the Solidity language[3] and of existing vulnerability taxonomies (Atzei et al., 2017; Dika and Nowostawski, 2018). This analysis aimed at identifying opportunities for designing new mutation operators (or improving the existing ones). Meaningful mutation operators should simulate likely faults that can manifest as anomalous contract behavior. The mutation should also be reasonably reachable from the test suite. Therefore, we focused on potential errors in the programmer's thought process and their effects on the contract execution.

Customized mutation process. SuMo allows the tester to select which mutation operators to apply and which contracts to mutate. By default, starting a mutation process runs all the implemented mutation operators. However, depending on the test suite and the contract under test, the developer might only want to apply specific operators so to reduce the overall cost. Indeed disabling selected operators can significantly reduce the number of generated mutants, and speed up the mutation testing process.

Limiting stillborn mutants. A stillborn mutant is generated when its mutation can be trivially detected by the compiler. Although test execution represents the major time-consuming factor of mutation analysis, stillborn mutants cause an unnecessary waste of CPU resources and contribute to slowing down the testing process. Since a full-scale mutation analysis can require hours to be completed on complex projects, focusing on the optimization of stillborn mutants can be ineffective. However, a substantial reduction in stillborn mutants can make a difference in the context of **incremental mutation testing**. In Section 6.1 we show how preventing compilation errors can positively impact incremental mutation analysis. Thus, we tackle the challenge of preventing compilation errors at different levels.

First of all, we identified already proposed operators that were empirically found to generate large amounts of stillborn mutants. The mutations that introduce a significant performance overhead without providing useful insights about the test suite were excluded from the set. Examples of such operators are the ones that target data types, the inheritance mechanism, and the pure and view modifiers.

However, it is not possible to completely prevent the generation of stillborn mutants with selective mutation strategies. Indeed, the insights provided by some mutation operators are too valuable, and they cannot be excluded without compromising the quality of the test suite. Therefore, we analyzed the Solidity language to identify mutations that might generate a compilation error. Based on this study, we designed corresponding **precondition checks** that must be applied during the mutant generation process. Some preconditions can be rather simple. For instance, we might ensure that a function is only mutated if it is not overridden. However, the mutation rules often rely on information collected during the visit of the AST to determine whether a mutation should be applied or not. For example, let us consider an operator that targets the return values of a function. Randomly swapping the return values can generate many stillborn mutants, since the values might belong to different data types. To prevent this from happening, SuMo visits the AST nodes containing the type associated with each return value and uses the information to perform legal replacements.

Reducing subsumed mutants. To mitigate the impact of subsumed mutants we combined two different techniques: **(1) Operator Optimizations** and **(2) Trivial Compiler Equivalence (TCE)**. SuMo allows the user to choose between two sets of mutation operators. The **Non-Optimized (Non-Opt)** operators include extended

[3] https://solidity.readthedocs.io/en/latest/index.html

Table 2
SuMo Solidity-specific mutation operators.

MutOp ID	MutOp Name	Aspect	Sec.
AVR	Address Value Replacement	*Address*	4.1.9
CCD	**Contract Constructor Deletion**	*Construction*	4.1.4
DLR	Data Location keyword Replacement	*Data Location*	4.1.11
DOD	Delete Operator Deletion	*Delete Keyword*	4.1.12
EED	Event Emission Deletion	*Event Emission*	4.1.5
EHC	Exception Handling statement Change	*Exception Handling*	4.1.13
ETR	Ether Transfer function Replacement	*Ether Transfer*	4.1.10
FVR	Function Visibility Replacement	*Visibility*	4.1.2
GVR	**Global Variable Replacement**	*Block and Transaction Properties*	4.1.7
MCR	**Mathematical and Cryptographic function Replacement**	*Global functions*	4.1.8
MOC	**Modifiers Order Change**	*Function behavior modifi.*	4.1.3
MOD	Modifier Deletion	*Function behavior modif.*	4.1.3
MOI	Modifier Insertion	*Function behavior modif.*	4.1.3
MOR	Modifier Replacement	*Function behavior modif.*	4.1.3
OMD	**Overridden Modifier Deletion**	*Function behavior modif.*	4.1.3
PKD	Payable Keyword Deletion	*Function state modifiers*	4.1.1
RSD	Return Statement Deletion	*Function state modifiers*	4.1.1
RVS	**Return Values Swap**	*Function state modifiers*	4.1.1
SCEC	Switch Call Expression Casting	*Address*	4.1.9
SFD	Selfdestruct Function Deletion	*Global functions*	4.1.8
SFI	Selfdestruct Function Insertion	*Global functions*	4.1.8
SFR	**SafeMath Function Replacement**	*Libraries*	4.1.14
TOR	Transaction Origin Replacement	*Block and Transaction Properties*	4.1.7
VUR	Variable Unit Replacement	*Units*	4.1.6
VVR	Variable Visibility Replacement	*Visibility*	4.1.2

mutation rules capable of generating a more comprehensive collection of mutants. However, the higher reliability guaranteed by this set comes at a price. Besides increasing the likelihood of generating redundant mutants, the Non-Optimized operators are responsible for a more expensive and time-consuming mutation testing process. Therefore, SuMo allows the tester to decide whether the mutation rules should be optimized before starting the testing process. The **Optimized (Opt)** operators consist of simplified rules that aim to limit the generation of likely redundant mutants. In particular, we merged those mutations that are likely to produce the same test outcome (in the scope of each mutation operator). The selection of the optimized rules was driven both by our knowledge of the Solidity language and by the analysis of existing mutation operators, such as the ones implemented in PIT.[4] The optimizations involve some specific mutation operators, which we better describe in Sections 4.1 and 4.2. The main drawback of the optimizations is the potential loss of some useful, non-subsumed mutants. Indeed, the subsumption relationships among the mutants do not have universal validity, as they strictly depend on the implementation of the test suite. In order to detect and remove subsumed mutants without risking information loss, we also employ Trivial Compiler Equivalence. As introduced in Section 2, TCE leverages compiler optimization technologies to detect functionally equivalent code. Besides being relatively fast and simple to apply, this technique has been extensively applied to real-world programs (Papadakis et al., 2015; Groce et al., 2018) with good results. TCE is applied right after the mutant generation phase, regardless of the employed operators.

Removing equivalent mutants. An equivalent mutant retains the same semantic of the original program. Therefore, it cannot be detected by any test suite. Equivalent mutants often elude automated detection techniques and must be weeded out manually. Nevertheless, Trivial Compiler Equivalence (Papadakis et al., 2015) represents one of the best solutions to this complex issue. Indeed, it can discard a significant amount (up to 30%) of equivalent mutations, reducing the manual effort required to the tester.

As a result SuMo currently implements a set of 25 Solidity-specific operators (see Table 2), and 19 general operators (see Table 3). In the tables, we highlighted in bold those mutation operators that either target features that were never considered before, or use updated mutation rules. In the following subsections, we present the two categories of operators separately. For those operators that foresee an optimized version, like AOR, BOR, ER, GVR, SFR, RVS, and VUR, we also report information about the simplified mutation rules. For the sake of readability the last column of the tables reports the subsection in which the operator is described. A comprehensive discussion on all implemented operators can be found on the SuMo page (http://pros.unicam.it/sumo).

4.1. Solidity-specific mutation operators

To identify mutation operators specifically targeting Solidity characteristics we considered 14 aspects of the language that could help in classifying and identifying useful operators. The classification is in line with the official Solidity documentation, though specific aspects were later refined or removed depending on our findings and the implemented mutation operators. As said this leads to the identification of 7 new Solidity mutation operators, while the remaining operators find some correspondence in existing works (Andesta et al., 2020; Chapman et al., 2019; Honig et al., 2019; Hartel and Schumi, 2020; Wu et al., 2019) even though many of them have been reconsidered, and slightly modified. SuMo also includes those operators that have been demonstrated to be effective, while mutation operators that generate a significant number of invalid mutants were excluded from the set. For instance, operators that target the inheritance mechanism might cause a child contract to show unexpected behaviors. However, altering the inheritance hierarchy prevents the contract from compiling most of the time. Mutation operators proposed in the literature that are no longer valid were either removed or updated to comply with the latest Solidity syntax. Several operators were also enhanced to simulate a wider range

[4] https://pitest.org/

Table 3
SuMo general mutation operators.

MutOp ID	MutOp Name	Aspect	Sec.
ACM	**Argument Change of overloaded Method call**	*Overloading*	4.2.6
AOR	Assignment Operator Replacement	*Expression*	4.2.1
BCRD	Break and Continue Replacement and Deletion	*Control*	4.2.2
BLR	Boolean Literal Replacement	*Literal*	4.2.3
BOR	Binary Operator Replacement	*Expression*	4.2.1
CBD	Catch Block Deletion	*Control*	4.2.2
CSC	Conditional Statement Change	*Control*	4.2.2
ECS	**Explicit Conversion to Smaller type**	*Type*	4.2.4
ER	**Enum Replacement**	*Type*	4.2.4
HLR	Hexadecimal Literal Replacement	*Literal*	4.2.3
ICM	Increments Mirror	*Expression*	4.2.1
ILR	Integer Literal Replacement	*Literal*	4.2.3
LSC	Loop Statement Change	*Control*	4.2.2
OLFD	**Overloaded Function Deletion**	*Overloading*	4.2.6
ORFD	Overridden Function Deletion	*Overriding*	4.2.5
SKD	Super Keyword Deletion	*Overriding*	4.2.5
SKI	Super Keyword Insertion	*Overriding*	4.2.5
SLR	String Literal Replacement	*Literal*	4.2.3
UORD	Unary Operator Replacement and Deletion	*Expression*	4.2.1

of faults. When possible, redundant mutations were merged to reduce the total number of generated mutants. The following subsections provide an analysis of the proposed Solidity-specific mutation operators and then implemented in the SuMo toolset.

4.1.1. Function state modifiers mutation operators

A function is a unit of code that can be executed within a contract as a result of a transaction or a message call. Solidity provides three standard modifiers (see Table 4) that can be added to a function to alter its behavior. Several tools implement operators for mutating the `pure` and `view` modifiers (Chapman et al., 2019; Wu et al., 2019). However, such mutations can generate a large number of stillborn mutants due to the possible disagreement between the specified behavior and the associated modifier. In addition, the erroneous usage of `pure` and `view` does not directly impact the behavior of the contract. The **PKD (Payable Keyword Deletion)** operator included in SuMo, differently from other proposals, just removes the `payable` modifier, if specified, to ensure that the affected function can correctly receive funds. This mutation is applied both to standard functions and to the fallback function. The fallback is a special function to be invoked in case a contract receives an invocation for which there is not a corresponding function implemented. However, it is also invoked on plain Ether transfers (e.g. via `send()` or `transfer()`) if the receive Ether function is missing. Virtual, overridden, and receive Ether functions are excluded to avoid compilation errors. Related works (Chapman et al., 2019) also propose mutations that add the `payable` keyword to functions. However, this error is highly unlikely to happen and translate into a failure. Moreover, it is only possible to kill this type of mutant if the tests attempt to send Ether to each contract function.

In Solidity, it is possible to return data from a function using two different types of syntax, and each function can have multiple return values. These peculiarities can be a source of confusion, and lead to mistakes in the implementation of a function. The **RSD (Return Statement Deletion)** operator included in SuMo aims to ensure the correct implementation of the return statement. The RSD operator removes the return statement from the body of each contract function, one at a time. This causes the affected function to return the default value for its return type (or types). RSD may cause the contract to show unexpected behavior, especially regarding the interaction between functions and the integration with other contracts. SuMo also implements the novel **RVS (Return Values Swap)** operator. In case a function returns multiple values, the RVS operator (Listing 5) swaps them among themselves. RVS targets both types of return structures supported by Solidity. The Non-Optimized version of RVS swaps all the compatible return values within the return statement. When the optimizations are enabled, each return value is swapped with a single compatible return value. In both cases, two values are only considered compatible if they belong to the same data type. This prevents the generation of stillborn mutants and speeds up the testing process.

Table 4
Function modifiers in Solidity.

Keyword	Description
`view`	The function cannot modify the state of the contract.
`pure`	The function can neither read nor modify the state of the contract.
`payable`	The function is allowed to receive Ether.

Listing 5: Return statement mutated by the RVS operator

```
1  //Original
2  function signatureSplit(bytes memory signatures, uint256
       pos)
3      internal pure returns (uint8 v, bytes32 r, bytes32 s) {
4          assembly {
5              let signaturePos := mul(0x41, pos)
6              r := mload(add(signatures, add(signaturePos, 0
                   x20)))
7              s := mload(add(signatures, add(signaturePos, 0
                   x40)))
8              ...
9          }
10 }
11
12 //Mutant
13 function signatureSplit(bytes memory signatures, uint256
       pos)
14     internal pure returns (uint8 v,  bytes32 s , bytes32 r ) {
15         assembly {
16             let signaturePos := mul(0x41, pos)
17             r := mload(add(signatures, add(signaturePos, 0
                   x20)))
18             s := mload(add(signatures, add(signaturePos, 0
                   x40)))
19             ...
20         }
21 }
```

Table 5
Function visibility keywords in Solidity.

Keyword	Description
`public`	The function can be accessed by all parties.
`external`	The function can only be accessed by external parties.
`internal`	The function can only be accessed by the current contract, and any contract derived from it.
`private`	The function can only be accessed by the current contract.

4.1.2. Visibility mutation operators

Solidity allows attaching visibility modifiers to functions (Table 5) and variables. These alter how the function (or variable) can be accessed by other contracts. Ensuring the correct usage of visibility keywords is essential to avoid the deployment of faulty code. Using a visibility keyword that is too restrictive might cause integration issues with other contracts. This type of fault does not necessarily lead to a software failure. However, depending on the internal logic of the affected function, an external party could be able to make unauthorized or unintended state changes. For instance, the affected function could perform critical operations such as withdrawing funds (*SWC-105* – SWC-n can be accessed at https://swcregistry.io/docs/SWC-n) or self-destructing the contract (*SWC-106*). State variables are permanently stored in the blockchain to maintain the current state of the contract. Functions can perform operations on these variables to read or update the contract state. SuMo implements two mutation operators that simulate faults concerning both function and variable visibility. The **FVR (Function Visibility Replacement)** operator replaces a function visibility keyword with a different one. There are some exceptions to this rule that we introduced to limit the possible generation of stillborn mutants. In particular, SuMo's implementation for FVR does not mutate the visibility for receive Ether functions and the fallback functions, as they can only be declared as external, and a mutation will lead to a stillborn mutant. Similarly, and for the same reason, it does not change the visibility of payable functions to internal or private. Lastly, FVR does not delete the visibility keyword, as the function default visibility is no longer allowed. The **VVR (Variable Visibility Replacement)** operator replaces the visibility keyword of a state variable with a different one. VVR also adds the private and public keywords to those variables that have the default visibility (which is equal to internal).

4.1.3. Function behavior modifiers mutation operators

Solidity permits the usage of additional modifiers, that can also be defined by the user. Similar to aspects in Aspect-Oriented Programming, modifiers permit to extend the behavior of a function with a predefined logic. This additional behavior is defined in a dedicated `modifier` function. Attaching a modifier to a function declaration can have different effects on its execution, although modifiers are most commonly used for implementing access control mechanisms. Errors in the usage of such kinds of modifiers can introduce faults of varying severity into the contract. For instance, attaching the wrong access modifier to a function might apply unnecessary restrictions to the transaction execution. This prevents the function from running every time the modifier conditions are not respected (*SWC-123*). Similarly, omitting a modifier might prevent a precondition check from taking place. In the worst-case scenario, this will lead to the unintended or malicious execution of critical operations, such as withdrawing funds or destroying the contract. The **MOD (Modifier Deletion)**, **MOI (Modifier Insertion)** and **MOR (Modifier Replacement)** operators implemented by SuMo simulate errors concerning the wrong usage of function modifiers. These operators were inspired by *Deviant* (Chapman et al., 2019) and *ContractMut* (Hartel and Schumi, 2020). In particular, the **MOD** operator (Listing 6) removes the modifier attached to each function signature, one at a time. This prevents the logic implemented by the modifier from running upon the execution of the function. The **MOI** operator applies an existing modifier to a function that does not have any. All the modifiers in the contract are evaluated to determine whether they can be attached to the target function. The number of mutations generated by MOI can be significant, therefore we defined several mutation preconditions to increment its efficacy. For instance, before applying a mutation, MOI checks whether the parameters of the modifier are compatible with the parameters of the target function. That is, the parameters of the modifier must be included in the list of parameters required by the function. This requirement helps to greatly reduce the number of stillborn mutants generated by MOI. The **MOR** operator, on the other hand, replaces the modifier attached to a function with a different modifier. This mutation can alter the restrictions applied to the function, and cause the execution of unexpected operations.

Listing 6: Function modifier mutated by the MOD operator

```
1  //Original
2  function updatePrice(uint newPrice) onlyOwner {
3    price = newPrice;
4  }
5
6  //Mutant
7  function updatePrice(uint newPrice) {
8    price = newPrice;
9  }
```

MOC (Modifier Order Change) is a novel operator that simulates the erroneous usage of multiple modifiers. If multiple modifiers are attached to a function signature, the MOC operator alters their order. As a result, the modifiers will be run in a different order upon the execution of the function. This mutation can have different effects on the contract, such as the function becoming unreachable. Lastly, **OMD (Overridden Modifier Deletion)** is a novel operator that targets the modifier overriding mechanism. OMD removes an overridden modifier from a derived contract. When the function decorated with the deleted modifier is called, it will be forced to use a modifier of a higher level in the inheritance hierarchy. Executing a different modifier version than the one intended can cause the contract to behave incorrectly. Moreover, the base modifier might apply different restrictions from the overridden one.

4.1.4. Construction mutation operators

A constructor is an optional function that is run upon contract creation. The constructor allows to initialize the state variables of a contract before the code is actually deployed on the blockchain. SuMo implements a novel **CCD (Contract Constructor Deletion)** mutation operator that targets the contract constructor. This operator was inspired by the JDC operator implemented by muJava (Ma et al., 2005; Ma and Offutt, 2014), which forces a Java class to execute the default constructor. The approach described by Andesta et al. (2020) is the only one that proposes dedicated mutation operators for altering a contract constructor. However, from Solidity version 0.4.22 the constructor is defined with the new `constructor` keyword, which requires the definition of a novel mutation. The CCD operator removes the constructor from a contract by commenting it out. As a result, the user-defined constructor will not be executed before deployment, and the contract state will not be correctly initialized. This mutation allows to determine whether the tests ensure the correctness of the contract state after deployment.

4.1.5. Events mutation operators

Solidity contracts can leverage the Ethereum Virtual Machine (EVM) logging facilities by firing events. Whenever an event is emitted its arguments are memorized in the transaction log, a special data structure that resides in the blockchain. Events are fired from a contract so that external parties can catch them and trigger a response. Even though the log is not accessible to contracts, events are often used within *DApps* (Decentralized Applications) to monitor the smart contract behavior and trigger code execution. The **EED (Event Emission Deletion)** operator (Listing 7) implemented by SuMo comments out an event emission statement to prevent its execution. This mutation encourages the definition of test cases that include conditions on events. SuMo does not include a mutation operator that swaps different event emissions within the contract. Indeed, this type of mutation can generate many redundant mutants. Moreover, since events often log the arguments of the function from which they are emitted, swapping event emissions has a high chance of generating a stillborn mutant.

Listing 7: Event mutated by the EED operator

```
1  //Original
2  function payout(address winner, uint256 amount) internal {
3      ...
4      emit Payout(winner, amount);
5  }
6
7  //Mutant
8  function payout(address winner, uint256 amount) internal {
9      ...
10     // emit Payout(winner, amount);
11 }
```

4.1.6. Variable units mutation operators

Solidity provides two variable units: Ether Units and Time Units. A literal number can take a suffix of `wei`, `gwei` or `ether` to specify a sub-denomination of Ether. Confusing the Ether units can have many dangerous side effects, such as transferring the wrong amount of funds. Using the wrong time unit can cause anomalous contract behavior as well. If the affected value is a trigger for performing actions, this fault can cause a piece of code not to execute, or to execute at the wrong time. SuMo implements the **VUR (Variable Unit Replacement)** mutation operator to simulate the incorrect usage of variable units. The VUR operator (Listing 8) replaces each ether and time unit suffix with a different, compatible suffix.

Listing 8: Ether unit mutated by the VUR operator

```
1  //Original
2  if(this.balance < 70 wei) {
3      uint amount = this.balance;
4  }
5
6  //Mutant
7  if(this.balance < 70 ether) {
8      uint amount = this.balance;
9  }
```

4.1.7. Block and transaction properties mutation operators

Solidity provides many special variables and functions to retrieve information about the blockchain. Confusing these variables can have unexpected effects on contract execution. Moreover, the improper usage of certain global variables, such as `block.timestamp`, can introduce dangerous dependencies into the contract. Variables that are subject to the miner's influence should never be used as a source of entropy (*SWC-120*), or to trigger time-dependent events (*SWC-116*). Solidity also provides global variables for retrieving the address of the account that triggered a transaction. While `msg.sender` identifies the sender of the current transaction call, `tx.origin` retrieves the root of the call chain. Confusing the two variables can cause a developer to obtain the wrong address. If the fault exists within an authorization check, it can result in unreachable code as well as unauthorized access (*SWC-115*). The novel **GVR (Global Variable Replacement)** operator included in SuMo was designed to avoid the incorrect usage of the global variables and functions available in Solidity. GVR allows simulating the manipulation of a global variable by a malicious miner, which can prevent the developer from inserting dangerous dependencies into the contract (such as using `block.timestamp` as a source of entropy). The GVR operator swaps each global variable (or function) in the contract with one or more different variables. In particular, the Non-Optimized version of GVR swaps all the compatible global variables with each other to maximize the variety of faults injected into the contract. Since this approach can generate a considerable amount of mutants, we also designed an Optimized version of GVR. The latter swaps those variables that are more likely to be confused during development, and only if they return a compatible data type. In particular, only the following replacements are performed:

- `(block.timestamp, {block.difficulty, block.number})`
- `(now, {block.difficulty, block.number})`
- `(block.number, {block.timestamp, block.difficulty})`
- `(block.difficulty, {block.timestamp, block.number})`
- `(block.coinbase,{tx.origin, msg.sender})`
- `(blockhash(), msg.sig)`
- `(block.gaslimit, {tx.gasprice, gasleft()})`
- `(gasleft(), {block.gaslimit, tx.gasprice})`
- `(tx.gasprice, {block.gaslimit, gasleft()})`
- `(msg.value, tx.gasprice)`

The **TOR (Transaction Origin Replacement)** operator is similar to GVR, but it targets a subset of global variables for retrieving the sender of a transaction. TOR swaps any instance of the `msg.sender` variable with `tx.origin`, and vice versa. If the transaction is part of a chain, this mutation causes the contract to obtain a different address than intended.

4.1.8. Global functions mutation operators

Solidity provides many global functions for performing different types of operations. These include the `addmod()` and `mulmod()` mathematical functions, which are used for modulo addition and multiplication respectively. Since they take in the same parameters, a developer could easily confuse them without noticing the error. Using the incorrect function leads to the computation of wrong values, as well as potential overflow conditions. Solidity also provides the `keccak256()`, `sha256()` and `ripemd160()` global cryptographic functions. These can be used for a variety of tasks, including hash calculation and public key retrieval. The novel **MCR (Mathematical and Cryptographic function Replacement)** operator implemented by SuMo replaces a mathematical or cryptographic function with a different function of the same type. The replacement rules for this operator are:

- `(addmod(), mulmod())`
- `(mulmod(), addmod())`
- `(keccak256(), sha256())`

- (sha256(), keccak256())
- (ripemd160(), sha256())

Solidity also provides a special `selfdestruct` function that allows to remove a smart contract from the blockchain. When executed, the `selfdestruct(address payable recipient)` method sends all the Ether stored in the contract to the designated address. The contract's data is then cleared from the state. The incorrect implementation of selfdestruct can be extremely dangerous, as there is no way of recovering the data or the Ether held by a destroyed contract. If the selfdestruct instruction is unreachable (or if it was omitted), it will not be possible to remove the contract from the blockchain. This is especially problematic if the deployed contract contains severe faults or vulnerabilities. The **SFD (Selfdestruct Function Deletion)** operator (Listing 9) included in SuMo comments out each selfdestruct function invocation, one at a time. This mutation prevents the contract from being eliminated from the blockchain. SFD helps to ensure the presence of tests that attempt to destroy the contract. The **SFI (Selfdestruct Function Insertion)** operator relocates a selfdestruct statement to the beginning of its function. This mutation aims to avoid access control measures and to introduce accidental selfdestruct invocations.

Listing 9: Selfdestruct function mutated by the SFD operator

```
1  //Original
2  function destroy() public onlyOwner {
3    selfdestruct(owner);
4  }
5
6  //Mutant
7  function destroy() public onlyOwner {
8    //selfdestruct(owner);
9  }
```

4.1.9. Address mutation operators

Solidity provides a special `address` type for handling contracts. Since addresses are required for performing a wide variety of operations, an address error can propagate in many unexpected ways. More in general, a call to the wrong address will either target a different contract than intended or a non-existent contract. If the faulty address exists, it is highly likely that the invoked function will not match any identifier of the recipient contract. This will either cause the transaction to fail, or the recipient fallback function to execute (*Call To the Unknown*). If the target address is orphaned, the transaction to the non-existent contract will fail. This is not a big issue in the case of normal function calls. However, sending Ether to an orphaned address results in the permanent loss of funds. The **AVR (Address Value Replacement)** operator included in SuMo helps to avoid mistakes regarding the usage of addresses. AVR targets hardcoded addresses, address identifiers and special address function calls such as `address(this)`. The mutation is applied both to address variable declarations and to simple address assignments. In particular, each address is always replaced with the contract's own address `address(this)`, and the zero address `address(0)`. When possible, it is also replaced with a different hardcoded address declared within the contract. SuMo also imports the **SCEC (Switch Call Expression Casting)** operator from Deviant. The SCEC operator swaps two address instances being cast to different contracts. This mutation makes the two contracts point to the incorrect address. Lastly, SuMo targets address members for transferring ether, which are discussed in Section 4.1.10.

4.1.10. Ether transfer mutation operators

Solidity provides several methods for transferring Ether between contracts: `transfer()`, `send()`, `call()`, `staticcall()` and `delegatecall()`. The first difference among these methods is the gas limit. When transferring Ether, it is possible to allocate a certain amount of gas for code execution. For `send()` and `transfer()` the stipend is limited to 2300 gas, which is just enough to perform simple operations. On the other hand, `call()` does not have a bounded gas limit, which means that a larger amount of gas can be drained to complete the execution of the recipient fallback. This can make the contract vulnerable to *Reentrancy (SWC-107)* attacks. The second difference is the return value in case of error. The methods that do not automatically revert the transaction upon failure require a manual check for the return value. If this is not handled properly by the developer, the exception will not be propagated any further. Lastly, Solidity provides two special variants of the message call. The `staticcall()` method is identical to a normal call, however, it does not allow any modification to the contract state. An erroneous usage of this method can prevent calls to library functions from working. The `delegatecall()` method dynamically loads code from a different address in the context of the current contract. This means that any modification to the state affects the storage of the caller, instead of the callee. A *delegate call to untrusted callee (SWC-112)* can be extremely dangerous, as it can cause unintended state changes. The **ETR (Ether Transfer function Replacement)** operator included in SuMo helps to avoid mistakes in the usage of ether transfer functions by replacing each ether transfer function with a different one. ETR is similar to operators included in other mutation testing proposals. However, in SuMo it was adapted to target the new syntax for the `call()` method, and it was augmented to mutate the `delegatecall()` and `staticcall()` methods. The replacement rules applied are:

- (send(), {transfer(), call()})
- (transfer(), {send(), call()})
- (call(), {delegatecall(), staticcall()})
- (delegatecall(), {call(), staticcall()})
- (staticcall(), {call(), delegatecall()})

4.1.11. Data location mutation operators

To every reference type in Solidity is associated a data location keyword that specifies where the variable is stored, where `memory` is used for storing temporary variables and `storage` is used for persistently storing state variables. Confusing a data location keyword can alter the behavior of the affected variable. Replacing `storage` with `memory` limits the lifetime of the variable to the execution of the function in which it is declared. Conversely, swapping `memory` with `storage` extends the lifetime of the variable. The gas cost is affected as well because storage variables are more expensive than memory variables. The **DLR (Data Location Replacement)** operator included in SuMo swaps the `memory` and `storage` data location keywords.

4.1.12. Delete mutation operator

The `delete` keyword is a Solidity-specific operator that assigns the default value to a variable. SuMo imports the **DOD (Delete Operator Deletion)** mutation operator from *MuSC* (Li et al., 2019). DOD removes any instance of the `delete` keyword from a contract, one at a time. This mutation prevents a variable from assuming its default value.

4.1.13. Exception handling mutation operators

Solidity manages exceptions with three different state-reverting functions. Since transactions are atomic, reverting all changes is the only way of preventing unsafe contract executions. When triggered, these functions revert all modifications to the state in the current call, including all sub-calls, and flag an error to the caller. `require()` is used to validate conditions that can only be checked at run-time. If the condition is not met, an exception is thrown and the unused gas is refunded to the sender. `assert()` is only used for invariants and internal error checking. Upon error, the transaction fails and the state changes are reverted without refunding gas to the sender. `revert()` is similar to require but it is used for handling business logic errors. Errors in the usage of exception-handling statements can have different effects. `require()` and `revert()` should not be used for internal error checking, while `assert()` should not be used for checking external inputs (*SWC-110*). A buggy or missing `assert()` statement does not ensure the proper validation of the contract state. Likewise, if a `require()` or `revert()` statement contains a fault, the run-time conditions will not be properly checked. The **EHC (Exception Handling statement Change)** operator (Listing 10) removes any instance of exception handling statement from a contract, one at a time. This mutation prevents the condition check from taking place. It also replaces an instance of exception handling statement with a different one. In particular, the run-time `require()` statement is swapped with `assert()` and vice versa. This mutation simulates the usage of an exception handling statement in the wrong context.

Listing 10: Revert statement mutated by the EHC operator

```
1  //Original
2  function buy(uint amount) public payable {
3    if (amount > msg.value / 2 ether)
4    revert("Not enough Ether provided.");
5  }
6
7  //Mutant
8  function buy(uint amount) public payable {
9    if (amount > msg.value / 2 ether)
10   //revert("Not enough Ether provided.");
11 }
```

4.1.14. Libraries mutation operators

SuMo implements the novel **SFR (SafeMath Function Replacement)** operator that targets the widely used SafeMath[5] library. SafeMath provides arithmetic functions that automatically revert the transaction in case of overflow. Developers often use them in place of the base arithmetic operators to increase code reliability. The SFR operator is available in two versions as well. The Non-Optimized version swaps each SafeMath function call with all different function call types. Such mutation rules permit to simulate the calculation of the wrong value. When the optimizations are enabled, SFR replaces each SafeMath function call with a single different one in the following manner:

- `(SafeMath.add(),SafeMath.sub())`
- `(SafeMath.sub(),SafeMath.add())`
- `(SafeMath.mul(),SafeMath.div())`
- `(SafeMath.div(),SafeMath.mul())`
- `(SafeMath.mod(),SafeMath.mul())`

[5] https://docs.openzeppelin.com/contracts/2.x/api/math

4.2. General mutation operators

This section illustrates the general mutation operators implemented by SuMo, which are summarized in Table 3. General mutation operators target traditional programming constructs that are not unique to the Solidity language. SuMo implements 19 general mutation operators, 4 of which are novel in the context of Solidity mutation. Several operators were inspired by well-documented tools such as muJava (Ma et al., 2005; Ma and Offutt, 2014, 2016) and PIT (https://pitest.org/), and adapted to the Solidity language. In particular, PIT's operators were considered because they allow limiting the generation of redundant and equivalent mutants.

4.2.1. Expression mutation operators

The operators supported by Solidity are similar to the ones available in JavaScript. SuMo implements several mutation operators that target Solidity expressions. Binary mutation operators target binary expressions, which consist of two operands and one operator. Unary mutation operators target unary expressions, which consist of one operand and one operator. Lastly, assignment mutation operators target the assignment of a value to a variable, which can either be another variable or a literal. **BOR (Binary Operator Replacement)** targets all binary arithmetic, relational, conditional, bitwise, and shift operators. The Non-Optimized version of BOR implements the same rules used by many related works: it swaps each instance of a binary operator with all the valid operators belonging to the same class. Since binary expressions are extremely common in any language, this approach tends to generate large amounts of mutants. Therefore, we allow the tester to use simplified mutation rules as well. While the Optimized operator might prevent the generation of some useful mutants, it can significantly speed up the testing process. Indeed, it replaces each binary operator with a maximum of two different operators of the same class. These mutations were designed on the basis of several operators proposed by PIT[6,7,8] which aim to limit both the number of equivalent mutants, and the number of total generated mutants. The replacement rules of Optimized BOR are:

- {(+, −), (−, +), (*,{/,**}), (**,*), (/,*), (%,*), (≪, ≫), (≫, ≪), (|,&), (&, |), (^,&)}
- {(&&, ||), (||,&&)}
- {(<,{<=, >=}), (>,{>=, <=}), (<=,{<, >}), (>=,{>, <}), (==,{! =}), (! =,{==})}

The **UORD (Unary Operator Replacement or Deletion)** mutation operator targets all unary arithmetic, bitwise and conditional operators. Each unary operator is either removed or replaced with a different operator of the same class. Unary expressions are subject to three types of mistakes: replacement, omission and insertion. However, related works such as MuSC (Wu et al., 2019) show that inserting unary operators generates elevate amounts of equivalent mutants. UORD deletes the logical NOT (`!`), bitwise NOT (∼) and unary − to simulate omission. The mutation rules that target increments (++) and decrements (−−) are based on PIT's Increment Mutator[9] and Remove Increments[10] operators. Swapping and removing the increment operators can be extremely useful since many contracts rely on their usage. The replacement rules of UORD are:

[6] https://pitest.org/quickstart/mutators/#MATH
[7] https://pitest.org/quickstart/mutators/#NEGATE_CONDITIONALS
[8] https://pitest.org/quickstart/mutators/#CONDITIONALS_BOUNDARY
[9] https://pitest.org/quickstart/mutators/#INCREMENTS
[10] https://pitest.org/quickstart/mutators/#REMOVE_INCREMENTS

- {(++,{−−, }), (−−,{++, }), (−,),(~,)}
- {(!,)}

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

- {(/ =, {∗ =, =}), (% =,{∗ =, =}), (+ =,{− =, =}), (− =,{+ =, =}), (∗ =,{/ =, =})
- (≫=,{≪=, =}), (≪=,{≫=, =})
- (^=,{& =, =}), (& =,{| =, =}), (| =,{&, =})}

Lastly, the **ICM (Increments Mirror)** operator replaces an increment (or decrement) shortcut assignment with its mirror. If a developer accidentally types `=-` instead of `-=`, the statement is interpreted in the wrong way. In fact, `=-` is an assignment followed by the unary -, while `-=` is a shortcut assignment which performs a decrement operation. The usage of this operator for Solidity was first introduced by Vertigo (Honig et al., 2019), which imported it from PIT. This allows to simulate a known typographical error which is also reported by the SWC-129. If a developer accidentally types `=+` instead of `+=`, the statement is interpreted in the wrong way. In fact, `=+` is an assignment followed by the unary +, while `+=` is a shortcut assignment which performs an increment operation. However, since the unary + is no longer supported by Solidity, the ICM operator was modified to only target decrements.

4.2.2. Control structures mutation operators

Solidity supports most of the known control structures. These are: `if`, `else`, `while`, `do`, `for`, `break` and `continue`. The `return` statement was discussed in detail in Section 4.1.1. Solidity also includes `try/catch` block for catching exceptions and reacting to the failure. A developer can introduce faults in conditional statements (`if` and `else`) whenever the condition is erroneously coded. This can depend on improper use of conditional and relational operators, as well as typographical errors concerning literals. Such errors cause the body to execute at the wrong time, or not to execute at all. Loop statements are subject to the same type of errors. Any fault in the condition of a `while` or `for` block affects the execution of the statements in the body. In Solidity a faulty loop statement could execute the wrong number of iterations and also lead to *out-of-gas* exception at run time. Lastly, the improper usage of the `break` and `continue` keywords causes an iteration to be skipped, or the loop to be terminated altogether. This can happen if a developer confuses the two statements, or inserts them at the wrong line of code. SuMo implements several mutation operators for control structures. The **BCRD (Break and Continue Replacement and Deletion)** operator swaps the `break` and `continue` statement within a loop. Replacing break with continue makes the loop skip one iteration instead of terminating immediately. Replacing continue with break entails the opposite effect. BCRD also removes every instance of break and continue, one at a time. This prevents the loop statement from working as intended. The **CBD (Catch Block Deletion)** operator removes one `catch` block from the `try/catch` statement. The deletion is only performed if there are at least two catch blocks, otherwise a compilation error is generated. When a catch block is removed, the operation to be performed upon exception will not be executed. For instance, this can be the logging of an error. The **CSC (Conditional Statement Change)** operator mutates conditional statements by replacing the entire condition with true or false. This causes the body to always execute, or to never be executed. CSC was augmented on the basis of the classic SDL (Statement Deletion) operator (Deng et al., 2013) to delete the `else` block as well. Lastly, the **LSC (Loop Statement Change)** operator sets the conditional statement within a `for` or `while` statement to true and to false. This mutation either generates either an infinite loop, or it prevents the body from running.

4.2.3. Literal mutation operators

Literal mutation operators replace hard-coded values within a contract with a different fixed value. In SuMo we included the **BLR (Boolean Literal Replacement)** operator that simply replaces true with false and false with true. The **ILR (Integer Literal Replacement)** operator replaces any integer literal with a different value. In particular, each value is incremented and decremented by one. The **HLR (Hexadecimal Literal Replacement)** operator replaces a hexadecimal literal with a zero or non zero hexadecimal value. The **SLR (String Literal Replacement)** operator replaces any string literal with the empty string.

4.2.4. Types and conversions mutation operators

Types and conversions mutation operators target data types that are not unique to Solidity. The novel **ER (Enum Replacement)** operator implemented by SuMo helps to ensure the correct usage of Enums. Enums are broadly used in smart contract development, as they allow the creation of user-defined types. Each Enum can contain one or more members, where the first member represents the default value. If a developer is unaware of the default Enum behavior, it might lead to wrong assumptions about the value stored in a variable. It is also possible that an Enum variable is mistakenly assigned the wrong value, which can cause unexpected contract behavior. Like other operators that perform keyword replacements, ER allows the tester to choose between extended and simplified mutation rules. By default, the ER operator (Listing 11) always swaps the first and second member of an Enum definition. This mutation forces the second members to become the default value of the Enum. The Non-Optimized version of ER replaces the member of an Enum assignment with each different member. If the optimizations are enabled, each member is only replaced with a single different member. Such mutation causes the affected Enum variable to assume a different value than intended. SuMo also implements the novel **ECS (Explicit Conversion to Smaller type)** operator that helps to avoid dangerous mistakes during explicit conversions of integer and byte types. ECS targets explicit integer conversions to simulate truncation faults. In particular, it forces a conversion to the smallest integer type to cause the loss of higher-order bits. Similarly, it forces explicit byte conversions to a smaller type than intended. This can cause the right-most bits to be truncated.

Listing 11: Enum definition mutated by the ER operator

```
1  //Original
2  enum StateType {
3    Created,
4    InUse
5  }
6
7  StateType public State;
8
9  function setState() public {
10   State = StateType.InUse;
11 }
12
13
```

```
14  //Mutant
15  enum StateType {
16    Created,
17    InUse
18  }
19
20  StateType public State;
21
22  function setState() public {
23    State = StateType.Created;
24  }
```

4.2.5. Overriding mutation operators

Solidity's overriding mechanism allows child contracts to redefine the behavior of certain derived functions, modifiers, and constructors. The overriding mechanism must be enabled on each public or internal function with two distinct keywords: A `virtual` function can be overridden by a derived contract, while an `override` function is overridden in the current contract. When an overridden method is called, the function of the same name (and parameter types) in the most derived contract is invoked by default. However, the contract name or the `super` keyword can be used to call functions in a different position within the inheritance hierarchy. Function modifiers and contract constructors can be overridden in the same manner, although they do not support the usage of the `super` keyword (or the contract name). SuMo implements several operators that target the overriding mechanism. The **ORFD (Overridden Function Deletion)** operator removes an overridden method from a derived contract. Any call to the deleted method will execute its base version instead. The main drawback of this mutation is that deleting a method overridden from multiple base contracts causes a compilation error. However, since multiple inheritance is not very common in smart contracts, this is a tolerable overhead. The **SKD (Super Keyword Deletion)** operator removes the `super` keyword from each function call in the contract, one at a time. Upon invocation, the contract will target the most derived version of the function instead of the parent's one. The **SKI** (Super Keyword Insertion) operator adds the `super` keyword to each overridden function invocation, one at a time. This mutation causes the call to execute a function of a higher level in the inheritance hierarchy. These operators were inspired by Deviant, which is the only other framework that implements overriding mutation operators. In addition, SuMo implements the **OMD (Overridden Modifier Deletion)** operator that targets the modifier overriding mechanism (Section 4.1.3).

4.2.6. Overloading mutation operators

Function overloading allows a smart contract to implement multiple functions with the same name, but with a different number (or type) of parameters. The most problematic aspect of overloading is that it can increase human confusion, which is the primary source of bugs. If a function is heavily overloaded, a developer could forget which version of the method does what. Moreover, when many versions of the same method are available, it is more likely to call an unintended method. Overusing overloading can also lead to the implementation of trivial methods which are not really needed. This can cause further confusion, as well as an increased amount of gas for contract deployment. SuMo includes two novel operators that target different aspects of method overloading in Solidity. These were inspired by the popular muJava (Ma et al., 2005; Ma and Offutt, 2014, 2016) framework. The **OLFD (Overloaded Function Deletion)** operator removes the declaration of each overloaded method within the contract, one at a time. This operator is useful in two ways. First, it allows to identify errors in the invocation of overloaded methods. If the mutant contract still works as expected without the deleted method, it means that such a method is not being called correctly. This can happen, for instance, when the developer invokes the wrong method version. OLFD also ensures coverage of overloaded methods. In fact, the test suite can only notice the mutation if the missing method is actually being called. The **ACM (Argument Change of Overloaded Method Call)** operator targets the invocation of overloaded methods within the contract. In particular, it swaps an overloaded method call with one different, existing overloaded method call. The mutated invocation must have a different number (or order) of arguments in order to match a different overloaded method. This causes a different method version to be invoked.

5. Design of SuMo

SuMo[11] is a mutation testing tool for smart contracts written in Solidity. The tool can automatically run a complete mutation testing process on a Solidity project, from mutant generation to test execution. The mutants are generated by 44 **mutation operators** that can be enabled and disabled by the user. As introduced in Section 4, certain operators come in two flavors: Optimized and Non-Optimized. Operator optimizations help to generate fewer mutants and speed up the mutation testing process, while disabling the optimizations guarantees a more thorough adequacy assessment. It is also possible to exclude selected contracts, such as libraries and contract migrations, from the mutation process. These features allow the developers to easily customize the mutation testing process to their needs and possibly reduce its cost. In Section 5.1 we provide an overview of the mutation testing process, while Section 5.2 focuses on the design and implementation aspects. Sections 5.3 and 5.4 provide details about the configuration of SuMo and the module used for detecting mutant equivalencies. Lastly, Section 5.5 describes the two versions of SuMo: (1) Stand-Alone tool and (2) Web Service.

5.1. Overview

Fig. 2 illustrates the flow of the mutation testing process conducted by SuMo. Upon testing, SuMo uses a Solidity parser[12] module to generate the AST (Abstract Syntax Tree) of each input SCUT (Smart Contract Under Test). Each AST is then visited to generate **mutations** according to the rules specified by each user-enabled operator. The resulting set of contracts can be passed to an external Java module, the Equivalent Mutant Detector (EMD), which applies the **Trivial Compiler Equivalence (TCE)** (see Section 5.4) to find mutant equivalencies. The redundant and equivalent mutants found by the EMD will be skipped by SuMo during the testing process. After applying a mutation to the corresponding contract, SuMo interacts with the underlying **testing environment** – which includes a testing framework (e.g., Truffle) and Ganache, an Ethereum blockchain simulator – to compile and test the mutated contract. Specifically, if the mutant can be compiled, SuMo runs the test suite provided by the developers to evaluate its effectiveness. Information about the mutation testing process, such as the number of generated stillborn mutants and the Mutation Score, is saved to a final **test report**. A copy of each generated mutation is also saved in a file. This allows the tester to examine the survivors for excluding the equivalent mutants and enhancing the test suite.

[11] SuMo page: http://pros.unicam.it/sumo

[12] Source Code https://github.com/ConsenSys/solidity-parser-antlr.

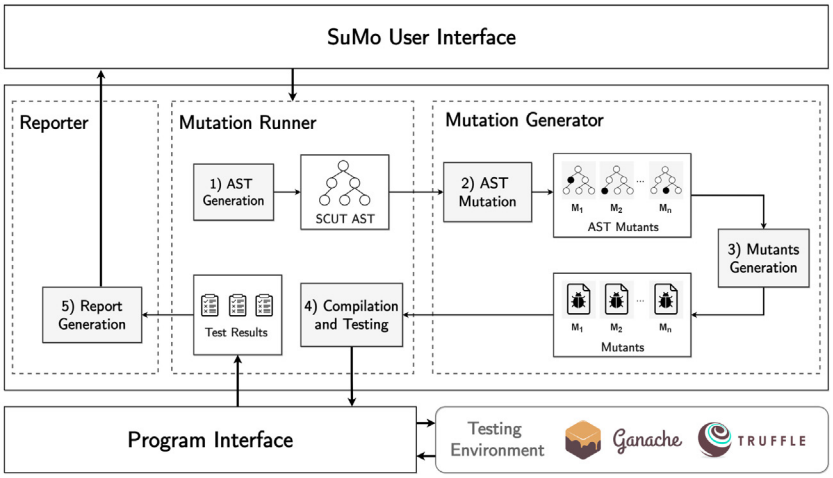

Fig. 2. Mutation testing process.

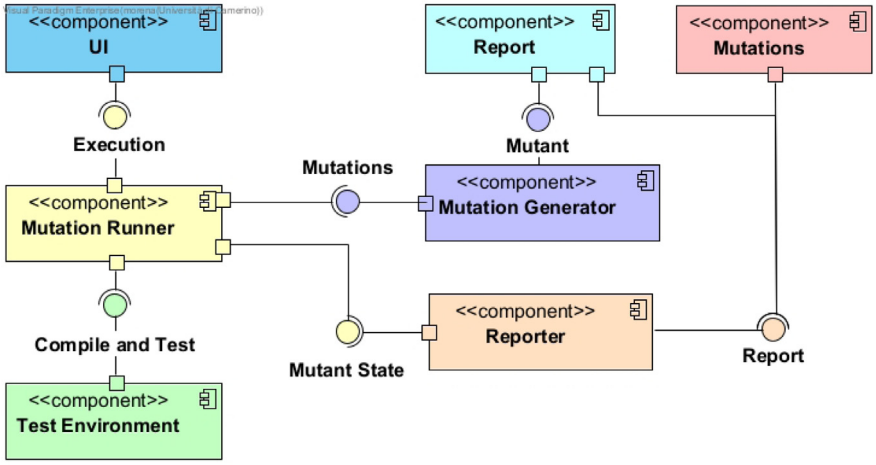

Fig. 3. Structure of SuMo.

5.2. Architecture

Fig. 3 shows the organization of the main logical components of SuMo, In the following paragraphs we describe the characteristics of each component, and we explain how a mutation testing campaign is executed. In particular, in the Mutation Testing paragraph, we describe how the core element of SuMo, the mutation runner, interacts with the other components to execute a mutation testing campaign.

User interface. The tester can interact with SuMo through a simple CLI (Command Line Interface). SuMo exposes several commands for managing the operators and for starting the mutation testing process. For instance, the `sumo preflight` command provides a quick overview of all the available mutations. By default, the `preflight` function generates a summary without applying the changes to the relative contracts. However, the configuration of SuMo can be overridden (see Section 5.3) to create and save each mutated contract to file. The `sumo test` command on the other hand, starts the actual mutation testing process. At the end of a mutation testing run, SuMo prints to screen a simple report that provides details about the status of each mutant. More detailed information is logged to file by the Reporter.

Program interface. The program interface allows SuMo to interact with the underlying testing environment, which must include a testing framework and a local blockchain network where the smart contracts can be deployed. SuMo was designed for running on top of the popular Truffle Suite,[13] which includes the Truffle testing framework and Ganache, an Ethereum simulator. Ganache permits setting up a local blockchain for development and testing purposes; developers can test their contracts in a safe environment that mimics the main chain, but without the delays introduced by the transaction validation process.

The execution of Ganache instances during the mutation testing process is managed by SuMo. The developers are only required to install and configure Ganache in their project (e.g., in their `truffle-config.js` file[14]), so that the testing framework (e.g., Truffle) can connect to it. On the other hand, the connection between SuMo and the testing framework must be established by the developers. Listing 12 shows a minimal example of the `spawnCompile()` function, which allows SuMo to check whether a mutant successfully compiles. Here, SuMo spawns a synchronous `compile` process in the root folder of the SUT. SuMo must know which `packageManager` to use for running this command (e.g., `npm` or `yarn`), which must be configured depending on the characteristics of the SUT (see Section 5.3). Additionally, the application developers are required to expose a `compile` script in the `package.json` of their application.

[13] Truffle Suite: https://www.trufflesuite.com/truffle.

[14] Truffle Configuration: https://trufflesuite.com/docs/truffle/reference/configuration.html.

Listing 12: Program Interface

```
function spawnCompile() {
  var child;
  if (process.platform === "win32") {
    compileChild = spawnSync(packageManager +
                ".cmd", ["compile"], {
        stdio: "inherit",
        cwd: projectDir
    });
  }
  ...
  return child.status === 0;
}
```

Overall, SuMo relies on the following two scripts that must be exposed by the SUT:

- `compile` compiles the smart contract(s) under test.
- `test` runs the test suite on the smart contract(s) under test.

These are quite standard scripts that are often present in real-world Ethereum projects. Although SuMo was designed for working with Truffle, it does not actually rely on a specific framework to compile and test the contracts; it connects to the one configured in the SUT (e.g., Hardhat[15]) Similarly, Ganache can be turned off (see Section 5.3) to allow different network configurations.

Mutation testing. As shown in Fig. 2, SuMo contains three main components: (1) Mutation Runner, (2) Mutation Generator and (3) Reporter. In the following, we describe each component in detail.

1. Mutation Runner: The mutation runner is the main component of SuMo. As illustrated in Fig. 4, the mutation runner interacts with the other components to provide the implemented functionalities to the user. In particular, it allows to manage and start the execution of a new mutation testing process. When the user launches a new mutation testing run, SuMo executes the following steps:

1. The mutation runner asks the mutation generator component to create the mutations.
2. The mutation generator instantiates visits for the AST of each smart contract that is the subject of the mutation testing activity. The mutations are identified based on the rules implemented by each enabled mutation operator.
3. The mutation generator saves preliminary information about the available mutations to the `report.txt` file.

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

4. The mutation runner checks whether the mutant should be discarded. To this end, it searches for the mutant ID within a list of redundant and equivalent mutants. Such list is provided by an external module that runs the TCE on the mutant binaries (see Section 5.4).
5. If the mutation is neither equivalent nor redundant, the mutation runner applies it to its contract.
6. The mutation runner starts a new Ganache instance, and attempts to compile the mutant through the underlying testing framework, such as Truffle.
7. If the compilation was successful, the mutation runner spawns a new test process. Otherwise, the mutant is reported as stillborn.
8. The mutation runner waits until the spawned process returns the test outcome. Since some mutations might induce infinite loops, SuMo sets an upper limit on the execution time of such process. The upper bound acts as a safeguard in the event that the user-defined test configuration does not specify any test timeout. By default, the tests are allowed to run for 10 minutes before the process is forcibly interrupted and the mutant is reported as timed-out. If the test process is completed, and at least one test failed, the mutant is considered killed.
9. The mutation runner kills the Ganache process, and restores the mutant contract to its original state so that the next mutation can be applied.
10. The mutation runner communicates the status of the mutant (stillborn, timed-out, alive, or killed) to the reporter.
11. If the mutant is valid, the reporter saves a copy of the mutation as a separate file.
12. If the mutation testing process is over, the reporter saves the Mutation Score and the status of each mutant to the `report.txt` file.

2. Mutation Generator: The mutation generator is responsible for the management and execution of the mutation operators. Each operator generates specific mutations by modifying a target contract according to its mutation rules. All the generated mutations are made available to the mutation runner, which is responsible for starting the actual testing process.

Each **mutation operator** takes as input the AST of the contract to be mutated, and it outputs a set of zero or more **mutations**. When the operator is run, it starts visiting one or more specific types of nodes within the AST. If the characteristics of the node match the mutation preconditions, the operator applies a syntactic change to the contract code. Each operator can generate multiple mutations per visit, depending on the mutation opportunities present in the contract under test. For instance, listing 13 shows the implementation of the Payable Keyword Deletion (PKD) operator. Since PKD aims to remove the payable keyword from specific contract functions, the visit method checks each `FunctionDefinition` node in the contract. However, the node is only mutated if the mutation preconditions are respected. In particular, the function must have `payable` state mutability, and it must not be marked as virtual, override, or the special receive Ether function. When a viable node is found, a new mutation is created by specifying an appropriate replacement value. In this case, the replacement consists of the function signature where the payable keyword has been replaced with the empty string. More complex operators visit the AST multiple times. This occurs when the operator must mutate different types of nodes, or when additional semantic information is needed to perform the mutation. Even though additional data is not always required, it is often useful to generate more precise mutations and to limit the generation of stillborn or redundant mutants.

Listing 13: Implementation of the PKD mutation operator

```
visit({
   FunctionDefinition: (node) => {
     var replacement;
     if(node.stateMutability === "payable" && !node
          .isReceiveEther && !node.isVirtual && !
          node.override) {
        var functionSignature = source.substring(
            node.range[0], node.range[1]);
        replacement = functionSignature.replace(
            "payable", "");
        mutations.push(new Mutation(file, node.
            range[0], node.range[1], replacement))
     }
   }
})
```

3. Reporter: The reporter component is responsible for showing and saving information about the ongoing mutation testing

15 Hardhat: https://hardhat.org.

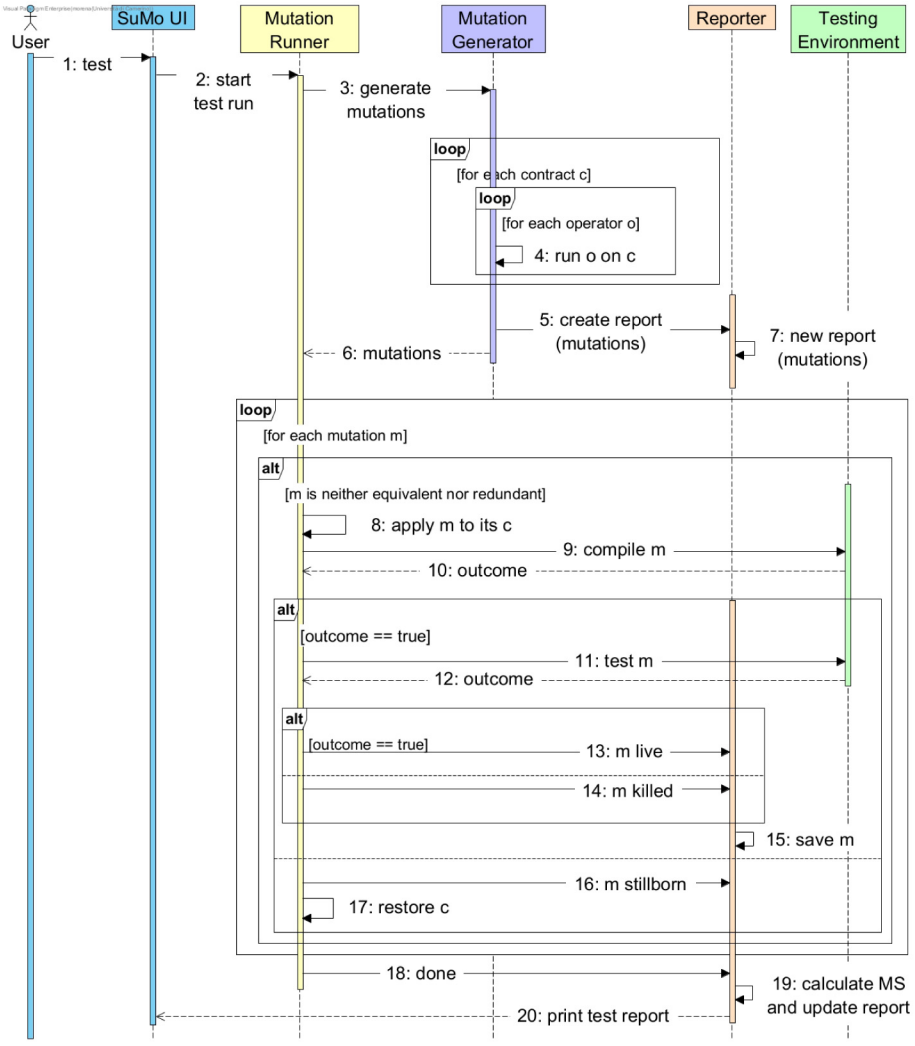

Fig. 4. Sequence diagram of SuMo.

process. The reporter takes as input the status of each mutant and writes a test report to the `report.txt` file. The report contains the list of mutations that were generated for each SCUT, as well as their generation and testing time. It also provides global information about the alive, killed, and stillborn mutants, and the Mutation Score achieved by the tests. The second type of artifact generated by the SuMo is the set of generated *mutations*. The Reporter saves each valid mutant to a separate file (see listing 14) so that it can be inspected at the end of the testing process.

Listing 14: Example of saved mutation

```
191 |  for (uint8 i = 0; i < player.hand.length; i++) {
192 |    uint8 card = (uint8) (player.hand[i] % 52 % 13);
193 | ---    score += cardValues[card];
    | +++    score -= cardValues[card];
194 |    if (card == 0) numberOfAces++;
195 |  }
```

5.3. Sumo configuration

SuMo contains a "`config.js`" file that allows the user to specify several options before starting the mutation testing process. Listing 15 shows an example of the configuration file used for running a given project. The the `packageManager` field is necessary to guarantee a successful integration with the underlying testing environment. Besides specifying the path to the root folder of the application under test, the user must also indicate the location of the contracts to be mutated. The `ignore` array contains an optional list of contracts that must be excluded from the mutation testing process. The `optimized` flag allows the tester to select whether to employ operator optimizations or not. The `ganache` flag allows the tester to choose whether SuMo should start Ganache or not during testing. Lastly, the `saveMutants` field can be enabled to save a copy of each contract mutated by SuMo. The main purpose of this option is to obtain a set of mutated contracts that can be passed to the Equivalent Mutant Detector.

Listing 15: SuMo configuration

```
module.exports = {
  packageManager: 'npm',
  ...
  projectDir:'C:/Users/username/projectDir',
  contractsDir:'C:/Users/username/projectDir/
      contracts',
  ignore: [
    'C:/Users/username/projectDir/contracts/
        Migrations.sol'
    ],
  optimized: true,
  ganache: true
```

Table 6
Experimental subjects.

DApp	Mutated Contracts	LOC	Test Suite Size	Test LOC	Stmt Cov.	Branch Cov.
Alice	28	970	201	1804	97,16	70,69
Blackjack	1	369	13	98	70,75	56,25
Coinosis	2	99	28	819	100	87,5
EtherCrowdfunding	1	428	35	902	93,83	71,3
Safe Contracts	21	1422	258	3531	95,23	99,73

```
11    saveMutants: true
12  }
```

5.4. Equivalent mutant detector

To improve the reliability and the cost-effectiveness of the proposed approach, we built an Equivalent Mutant Detector that uses Trivial Compiler Equivalence to prune the redundant and equivalent mutants generated by SuMo. The Equivalent Mutant Detector is a Java module that relies on the optimizations performed by the Solidity compiler to find equivalences between contracts. The program takes as input a set of contracts and produces a list of equivalent or redundant mutants that should be discarded. When the optimization mechanism of the compiler is enabled, two syntactically different contracts can end up with the same binary code. In particular, for the compilation phase, we relied on the following command: `solc sourcePath --metadata-hash none --optimize --bin -o destinationPath`. Therefore, the Equivalent Mutant Detector simply compares a set of binaries to detect trivially equivalent mutants. To discard eventual duplicates the Equivalent Mutant Detector compares all the mutants generated from the same contract.

5.5. SuMo versions

SuMo is currently available as a stand-alone program that can be downloaded locally. However, we are also providing the implementation of SuMo as a web service. Both the SuMo stand-alone and the SuMo web service implement the same core functionalities.

SuMo stand-alone tool. The stand-alone version of SuMo provides a Command Line Interface (CLI) for managing the mutation testing process and selecting the mutation operators to apply. This version makes few assumptions about the environment and the SUT (Software Under Test), therefore it is more suitable for projects that require complex and/or customized setups. In this case, the framework requires access to a locally installed version of the SUT. A dedicated configuration file allows the user to specify which smart contracts must be ignored during the mutation process. SuMo interacts with the SUT through the conventional "test" and "compile" scripts, which must be specified in the `package.json` file of the project. These are the scripts that SuMo invokes for running the tests and for compiling the mutated contracts respectively.

SuMo web service. On the other hand, the SuMo web service comes with a built-in REST API that offers the possibility of starting and managing an automatic mutation testing process with minimal effort. In this case, the configuration file and the target project, comprising the contract files, the test suite, and all the relative dependencies, can be directly uploaded to the SuMo server. All the functionalities concerning the execution of the tests, and the management of the mutation operators, are made available through the implemented API layer. The user can select which mutation operators to apply, launch the mutation testing process, and receive a test report upon completion. However, to guarantee the automation of the process, the SUT uploaded to the SuMo server must comply with specific requirements concerning the configuration of the test net and the test environment.

6. Experimental evaluation

To evaluate the effectiveness of our approach we selected several open-source DApps from GitHub (see Tables 6 and 7). In particular, we chose applications showing some complexity and with different characteristics, so to have the possibility to check a wider range of Solidity features. At the same time we looked for projects with a high-coverage test suite, so as to ensure some investment in testing. The applications we could retrieve and experiment with are:

1. **Alice**[16]: Alice is a platform built on Ethereum that permits to create communities with a focus on generating some social and individuals benefit (https://www.aliceDApp.com/).
2. **Blackjack**[17]: Blackjack is a DApp that permits to run a Blackjack game on Ethereum.
3. **Coinosis**[18]: Coinosis is a collaborative learning DApp that promotes the creation of study groups on various topics. People with common interests can join the group, and be rewarded for sharing their knowledge according to the recognition of the participants.
4. **EtherCrowdfunding**[19]: EtherCrowdfunding is a DApp for the organization of crowdfunding campaigns. It supports the creation of a new campaign, the donation of funds from the donors, and the reception of funds from the beneficiaries.
5. **Safe Contracts**[20]: The Gnosis Safe is a smart contract wallet with multi-signature functionality that allows secure management of blockchain assets.

Fig. 5 depicts the adopted experimental procedure. To validate the effectiveness of the proposed approach, we performed two separate mutation testing runs on each case study. During the first run, we turned off operator optimizations to assess the efficacy of the extended mutation rules. The second run, on the other hand, aimed to evaluate the impact of the optimized operators. The mutation testing runs comprise the following steps:

1. We mutate the original set of contracts I with either the non-optimized (*Non-Opt*) or the optimized (*Opt*) operators to generate a set of non-optimized (M_n) or optimized (M_o) mutants, respectively.
2. We compile each mutant $m \in M_n$ (or each $m \in M_o$) and each contract $i \in I$. The resulting sets of binaries C_n for the sources $m \in M_n$ (C_o for $m \in M_o$) and C_i for the are required by the Trivial Compiler Equivalence to detect program equivalencies.

16 https://github.com/alice-si/alice-v1-monorepo
17 https://github.com/0x9060/blackjack
18 https://github.com/coinosis/coinosis
19 https://github.com/giobart/EtherCrowdfunding
20 https://github.com/gnosis/safe-contracts

Table 7
Mutated contracts of the subject applications.

DApp	Mutated Contracts		
Alice	AliceToken.sol	DonationWallet.sol	Privileged.sol
	ClaimsRegistry.sol	Escapable.sol	Project.sol
	ConditionalCoupon.sol	FlexibleImpactLinker.sol	ProjectCatalog.sol
	Coupon.sol	ImpactLinker.sol	ProjectWithBonds.sol
	CuratedTransfers.sol	ImpactRegistry.sol	SimpleContractRegistry.sol
	CuratedWithWarnings.sol	InvestmentWallet.sol	StringUtils.sol
	DemoInvestmentWallet.sol	MockValidation.sol	TwoPhaseTransfers.sol
	DemoToken.sol	MoratoriumTransfers.sol	Vault.sol
	DigitalEURToken.sol	OffChainImpactLinker.sol	
	DigitalGBPToken.sol	OwnableWithRecovery.sol	
Blackjack	Blackjack.sol		
Coinosis	Event.sol	ProxyEvent.sol	
EtherCrowdfunding	CrowdfundingCampaign.sol		
Safe Contracts	CompatibilityFallbackHandler.sol	GnosisSafeL2.sol	SecuredTokenTransfer.sol
	DefaultCallbackHandler.sol	GnosisSafeProxy.sol	SelfAuthorized.sol
	Enum.sol	GnosisSafeProxyFactory.sol	SignatureDecoder.sol
	EtherPaymentFallback.sol	GuardManager.sol	SimulateTxAccessor.sol
	Executor.sol	HandlerContext.sol	Singleton.sol
	FallbackManager.sol	IProxyCreationCallback.sol	StorageAccessible.sol
	GnosisSafe.sol	ModuleManager.sol	
		OwnerManager.sol	

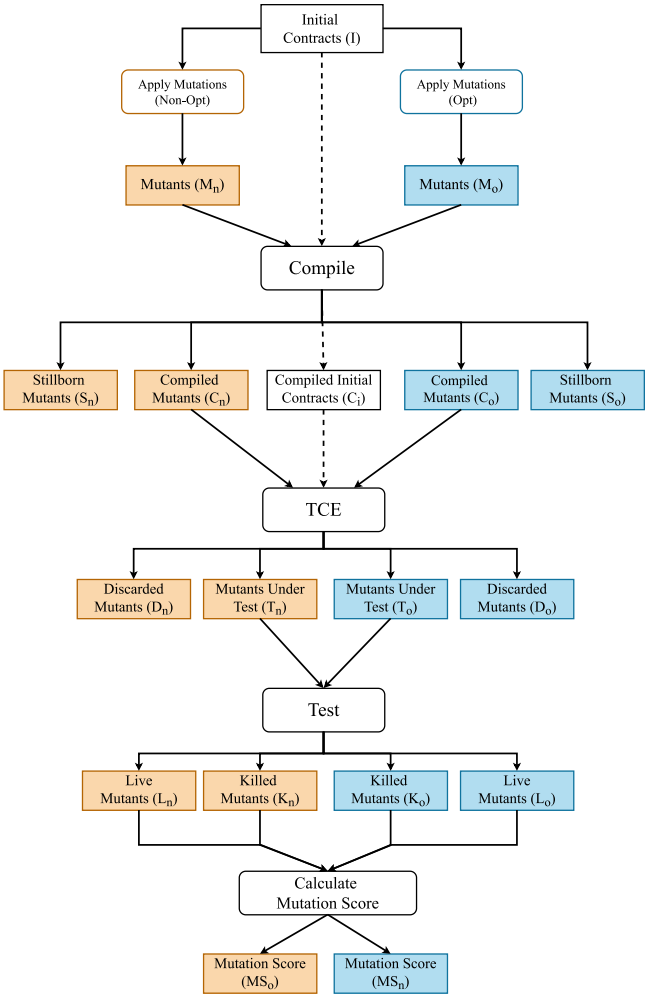

Fig. 5. Experimental process.

3. We apply the TCE on the aforementioned sets of binaries. If we chose to generate M_n, the Equivalent Mutant Detector (see Section 5.4) performs comparisons between the relative contract binaries. Then, it outputs two different artifacts: D_n is the set of mutants discarded with the TCE, while the set of mutants under test $T_n = C_n - D_n$ contains the contracts that must be tested by SuMo. Note that this set might still contain some undetected equivalent mutants, and is not necessarily equal to the set of valid mutants. The process stays the same whether we chose to use operator optimizations or not. If the optimizations are employed, the TCE outputs a set of mutants under test $T_o \subseteq T_n$.
4. We test each mutant $m \in T_n$ (or each $m \in T_o$) with SuMo. Depending on the outcome of the testing process, m can either be marked as killed or live (see Section 5). The subset of killed mutants K_n (or K_o) permits to calculate the Mutation Score MS_n (or MS_o) achieved by the test suite.

6.1. Experimental results

In this section we provide a general discussion of the experimental results, whereas in Sections 6.2–6.4 we answer the research questions posed in Section 3.

Tables 8(1) and 8(2) provide the results for the mutation testing run carried out with the non-optimized and the optimized operators, respectively. For each DApp, the two tables show the total number of generated mutants (M_n and M_o), stillborn mutants (S_n and S_o), the compiled mutants (C_n and C_o), the mutants discarded with the Trivial Compiler Equivalence (D_n and D_o), the mutants under test (T_n and T_o), the killed mutants (K_n and K_o), and the live mutants (L_n and L_o). The last column shows the Mutation Score achieved by the relative test suite, for the **non-optimized run** and the **optimized run**, respectively. Moreover, for each DApp the first line reports the data when all the mutation operators were used, the second line just focuses on Solidity-related operators, while the third one reports the observed values for the new operators introduced in SuMo.

As can be seen from Table 8, the test suites shipped with the selected applications were unable to kill a considerable amount of mutants, both during the non-optimized and the optimized

Table 8
Experimental Results — (1) Non-Optimized vs. (2) Optimized runs.

(1) **Non-optimized**									
DApp	Mutation Operators	M_n	S_n	C_n	D_n	T_n	K_n	L_n	MS_n (%)
Alice	All	1296	153	1143	68	1075	647	428	60,2
	Solidity	799	151	648	54	594	334	260	56,2
	SuMo	56	15	41	0	41	20	21	48,8
Blackjack	All	877	0	877	74	803	291	513	36,2
	Solidity	265	0	265	36	229	94	136	41
	SuMo	159	0	159	3	156	77	79	49,4
Coinosis	All	231	0	231	29	202	167	35	82,7
	Solidity	106	0	106	24	82	63	19	76,8
	SuMo	26	0	26	0	26	26	0	100
Ether Crowdfunding	All	899	2	897	108	789	540	249	68,4
	Solidity	425	0	425	68	357	206	151	57,7
	SuMo	140	0	140	1	139	129	10	92,8
Safe Contracts	All	1424	135	1289	376	913	699	214	76,6
	Solidity	626	97	529	169	360	268	92	74,4
	SuMo	130	17	113	12	101	71	30	70,3
Total	All	4727	290	4437	655	3782	2343	1439	**61,9**
(2) **Optimized**									
DApp	Mutation Operators	M_o	S_o	C_o	D_o	T_o	K_o	L_o	MS_o (%)
Alice	All	1036	147	889	59	830	491	339	59,2
	Solidity	775	145	630	54	576	328	248	56,9
	SuMo	32	9	23	0	23	14	9	60,9
Blackjack	All	536	0	536	56	480	189	291	39,6
	Solidity	213	0	213	36	177	76	101	43,5
	SuMo	65	0	65	1	64	34	30	53,1
Coinosis	All	156	0	156	26	130	103	27	79,2
	Solidity	94	0	94	24	70	51	19	72,8
	SuMo	14	0	14	0	14	14	0	100
Ether Crowdfunding	All	600	0	600	97	503	323	180	64,2
	Solidity	370	0	370	68	302	151	151	50
	SuMo	70	0	70	1	69	60	9	86,9
Safe Contracts	All	971	127	844	259	585	448	137	76,6
	Solidity	557	89	468	164	304	229	75	75,3
	SuMo	61	9	52	7	45	32	13	71,1
Total	All	3299	274	3025	497	2528	1554	974	**61,5**

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

run; the test suites achieved a global MS_n = 61,9% and MS_o = 61,5%. The adequacy further decreases if we only consider Solidity-specific operators. Not surprisingly, Solidity-specific mutants seem more difficult to kill and this could somehow be due to the fact that programmers need to get more acquainted with the novel language mechanisms. Furthermore, ∼10% of the non-optimized live mutants, and ∼6% of the optimized ones, were generated by the SuMo-specific operators.

Fig. 6 compares the adequacy value of each test suite in terms of its (1) statement coverage, (2) branch coverage, (3) Mutation Score (optimized), and (4) Mutation Score (non-optimized). As can be seen, the Mutation Score and the code coverage values convey a different perception of the test suites' quality; the Mutation Score is consistently lower for each selected case study. Particularly interesting is the case of Safe-Contracts that, notwithstanding a ∼ 100% statement and branch coverage, only reached a Mutation Score of ∼ 76% during both mutation runs. Overall, these results confirm that test suites with high values in the assessment of structural coverage parameters do not necessarily ensure the reliability of the smart contract code, as they can still miss a large number of faults.

A detailed discussion on the effects of each mutation operator is reported in Section 6.2.

Stillborn mutants. The percentage of stillborn mutants generated by SuMo is relatively small, which suggests the effectiveness of the implemented precondition checks. In particular, we can observe that only 8,3% of the optimized mutants and 6,13% of the non-optimized mutants were marked as stillborn. Compilation errors are unevenly distributed among the operators, and they seem to be more frequent when dealing with Solidity-specific mutations. As we better discuss in Section 6.2, we pinpointed three mutation operators that play a major role in the generation of stillborn mutants. DLR (Data Location Replacement) and ETR (Ether Transfer Replacement) feature the highest stillborn mutant

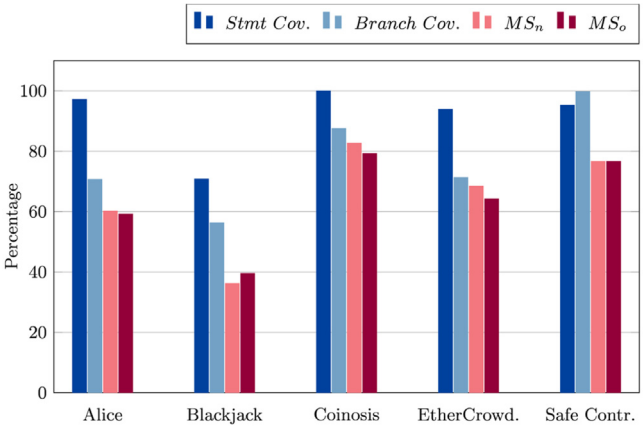

Fig. 6. Adequacy of the test suites shipped with the selected applications.

Table 9
Overhead induced by a stillborn mutant.

DApp	Avg. Time Wasted (s)
Alice	7,59
Coinosis	5,92
Blackjack	6
EtherCrowdfunding	8,9
Safe Contracts	7,2
Avg.	7,12

generation rate, while the largest amount of stillborn mutants was produced by FVR. This information can help testers to adapt the mutation testing process to their needs, so to reduce the costs associated with the compilation phase. As introduced in Section 4, limiting the generation of stillborn mutants can make a difference in the context of *incremental mutation testing*. In order to provide some concrete evidence of the slowdown induced by these mutants, we carried out an additional experiment. First, we modified the rules of a mutation operator to force the injection of a compilation error. The choice fell on the CCD (Contract Constructor Deletion) operator because it addresses a common Solidity feature observed in all of the selected DApps. Then, we ran CCD on each project. Table 9 reports the average time wasted by a single stillborn mutant on 20 independent runs (Col. 2). Such values include the time required for applying the mutation to the contract and the time wasted for attempting its compilation. The data shows that the average slowdown induced by a single stillborn mutant is around 7 seconds. This value can vary depending on the characteristics of the contract under test. For instance, a stillborn `CrowdfundingCampaign.sol` mutant from the EtherCrowdfunding project can waste almost 9 seconds. In this case, preventing the generation of 20 stillborns translates into an optimization of around 3 minutes. Therefore, focusing on the reduction of stillborn mutants can help us to build a more solid base for continuous integration practices where test suites are continuously modified and assessed, while it does not certainly constitute a major issue in full-scale testing activities.

Equivalent and redundant mutants. Column 5 shows that the TCE detected a significant amount of program equivalencies both during the optimized and the non-optimized test runs. In particular, $|D_n| = 655$ and $|D_o| = 497$. This means that 13,8% of the mutants in M_n and 15% of the mutants in M_o were marked as either equivalent or redundant. Thus, applying TCE contributed to significantly decreasing the cost of the testing process, and obtaining a more reliable Mutation Score. Surprisingly, most equivalencies were detected among the Solidity-specific mutants. Particularly interesting here is the case of the Safe-Contracts DApp. Indeed, almost 30% of its Solidity mutants were discarded before testing. The SuMo-specific operators, on the other hand, were seldom marked as equivalent or redundant.

6.2. Impact of the mutation operators

In this section we discuss the results achieved by the novel and the Solidity-specific mutation operators to answer RQ1: "*Do the SuMo-specific and Solidity-specific operators contribute to the identification of weaknesses in smart contract test suites?*". As shown below, the Solidity-specific operators contribute with a relevant fraction to the total number of live mutants, both in their Optimized (Fig. 7) and in their Non-Optimized (Fig. 8) version. Except for the Blackjack project, the Solidity mutations consistently survived more times with respect to the general ones. During the optimized runs, the Solidity-specific operators achieved on average a 58, 4% Mutation Score, whereas the general ones reached

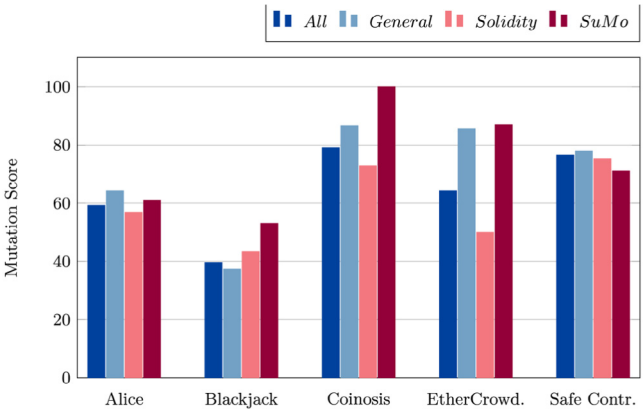

Fig. 7. Mutation Score achieved by the projects (Optimized).

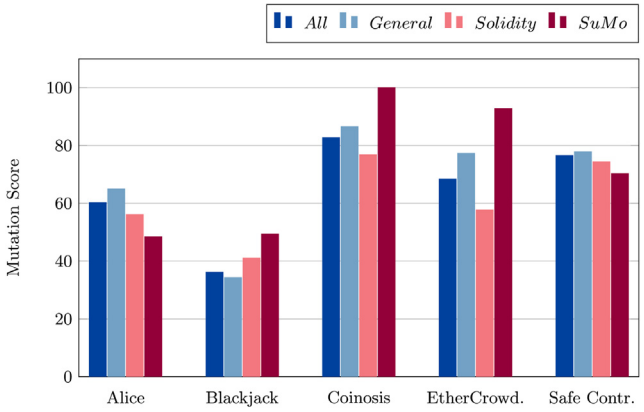

Fig. 8. Mutation Score achieved by the projects (Non-Optimized).

a 65, 4% Mutation Score. The non-optimized runs show a similar trend, with a 59, 4% Solidity-specific Mutation Score and a 63, 8% general Mutation Score. The Mutation Score of the SuMo-specific operators (i.e., operators firstly introduced by SuMo), was generally higher throughout the experiments, with the exception of the Safe-Contracts project. The average SuMo-specific Mutation Score is 69, 8% for the non-optimized runs, and 71, 6% for the optimized runs. Overall, the SuMo-specific mutations do not seem more difficult to kill, however they simulate both traditional and Solidity-specific faults.

Table 10 shows the results for the non-optimized mutation operators applied during the experiments, while Table 11 provides additional details about the mutants generated by means of the optimized rules. For each operator (Col. 1), the tables describe the total number of generated mutants (Col. 2), the number of stillborn mutants (Col. 3), the number of mutants discarded with the Trivial Compiler Equivalence (Col. 4), the number of mutants under test (Col. 5) and the achieved Mutation Score (Col. 6). The experiments allowed us to identify which classes of mutants are less likely to be killed by the tests. Even though we were not able to analyze in detail each live mutant, we can still make observations relative to type of injected fault and the potential usefulness of each operator.

AVR — Address Value Replacement. The **AVR** mutants were often killed by the provided test suites. The effects of an address mutation on the contract execution can vary largely. For instance, performing an incorrect address assignment might lead to the retrieval of the wrong balance value. In any case, altering an address is likely to have a drastic effect on the behavior of the

Table 10
Non-Optimized mutation operators results.

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

Table 11
Optimized mutation operators results.

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

Legend: M = total mutants, S = stillborn mutants, D = equivalent and redundant mutants discarded by the TCE, T = mutants under test, MS = Mutation Score.

contract. Our analysis of the live mutants suggests that the AVR operator can expose a wide range of issues with a given test suite. For instance, an AVR survivor might indicate that the contract state is not properly checked after an address assignment. Several AVR survivors were also useful to spot a lack of coverage for entire methods as well as guard clauses that include conditions on addresses.

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

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

DOD — Delete Operator Deletion. The **DOD** operator only generated 1 mutant under test. This is not surprising as the `delete` statement is not as frequent as other operators in real-world smart contracts. The survival of this mutant shows a weakness in the test suite: the state of the contract is not properly checked after the affected variable is set to its default value.

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

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

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

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

ETR — Ether Transfer Replacement. The **ETR** operator generated 8 mutants under test, none of which were killed by the tests. This result is not surprising because swapping ether transfer functions does not always affect the contract execution. In fact, the main difference between `send`, `transfer` and `call` is how they behave upon error. Given our results, it is safe to assume that testers rarely simulate a faulty ether transfer. Indeed, designing dedicated test scenarios that force each transfer operation to fail might not always be feasible, as it can require too much effort. However, considering that a Solidity project typically contains a small number of transfer operations, the usage of ETR can be considered.

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

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

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

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

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

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

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

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

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

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

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

> **Answer to RQ1:** The experiment we run seems to justify the introduction of the novel operators, and the usage of Solidity-specific operators, which can contribute with a relevant fraction to the total number of undetected mutants.

6.3. Operator optimizations

To answer RQ3 we analyze the experimental results, comparing the non-optimized and the optimized runs in terms of their Mutation Score, unique live mutants, subsumed mutants, and time costs. As can be seen from Table 8, both the non-optimized and the optimized operators convey a similar perception of the test suites quality. Indeed, the two test runs resulted in almost identical Mutation Scores. From Table 8-1 we can see that the provided test suites achieved a global $MS_n = 61{,}9\%$ on a total of 3782 mutants. The optimized run (Table 8-2) shows a similar outcome with $MS_o = 61{,}5\%$ on 2528 tested mutants. Each assessed test suite consistently killed a higher percentage of mutants during the non-optimized run. Nevertheless, employing the *Non-Opt* set can help to derive a more precise indication of the test suite quality, and to guarantee higher reliability standards. Indeed, the set of live mutants L_n can provide more valuable information for improving the test cases.

Table 12 shows the difference between the number of mutants tested during the non-optimized and the optimized test runs (Col. 2) for each project (Col. 1). It also shows how many live mutants can only be generated when the non-optimized operators are employed (col. 3). In particular, from Column 2 we can see that $|T_n - T_o| = 1254$. This is the total number of potentially useful mutants that cannot be generated when the optimizations are enabled. Of these 1254 mutants, 456 survived testing. This number suggests that the non-optimized operators can provide additional information about the test suite quality, however at the cost of a more expensive testing phase. Indeed the non-optimized run generated in average one and half more mutants with respect to the optimized one. Clearly, further studies are needed to determine whether T_n contains equivalencies that the TCE was not able to detect and if improvements to the test suites still bring similar scores for the two runs.

Table 12
Difference between Non-Optimized and Optimized operators.

DApp	$T_n - T_o$	$L_n - L_o$
Alice	245	89
Blackjack	323	222
Coinosis	72	8
EtherCrowdfunding	286	69
Safe Contracts	328	77
Tot.	1254	465

To further assess the impact of the operator optimizations, we analyzed the subsumption relationships present between the tested mutants. We recall that, according to the definition proposed by Kurtz et al. (2014), one mutant $m1$ is subsumed by another mutant $m2$ if at least one test kills $m2$ and every test that kills $m2$ also kills $m1$. Because there are no restrictions on the test set, we cannot compute the "true" subsumption; we must approximate the subsumption relationships based on mutant behavior against the specific test suite of each project. To identify the subsumption relationships and derive the number of subsumed mutants, we performed the following steps at the end of each mutation testing run:

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

Considering the subsumption relationships computed for each project, Table 13 shows the number of subsumed mutants $\mathbf{Su_n}$ generated by the Non-Optimized mutation operators (Col. 3), and the number of subsumed mutants $\mathbf{Su_o}$ generated by the Optimized mutation operators (Col. 4). The data shows that enabling operator optimizations helps to significantly reduce the number of subsumed mutants under test. The BOR, GVR, and SFR operators seem to benefit the most from the simplified set of rules. In the case of the Safe-Contracts project, the Non-Optimized BOR operator generated 180 subsumed mutants, while its optimized version only generated 9. Clearly, we must perform a deeper analysis to establish how many useful mutants could be lost due to operator optimizations. Indeed, both the number of useful and the number of subsumed mutants that can be removed with the optimizations can change depending on the implementation of the test suite. In general, it is not possible to select a subset of replacement rules that achieve perfect results for every project.

Table 13
Subsumed mutants.

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

Legend: *AOR* = Assignment Operator Replacement, *AVR* = Addres Value Replacement, *BOR* = Binary Operator Replacement, *EHC* = Exception Handling Change, *FVR* = Function Visibility Replacement, *GVR* = Global Variable Replacement, *ILR* = Integer Literal Replacement, *LSC* = Loop Statement Change, *SFR* = Safemath Function Replacement, *UORD* = Unary Operator Replacement and Deletion, *VVR* = Variable Visibility Replacement.

Table 14
Opportunities for operator optimizations.

MutOp	Subsuming rule	Subsumed rule	Projects
FVR	Public → internal	Public → private	4/5
FVR	External → internal	External → private	3/5
VVR	Public → internal	Public → private	3/5

Table 15
Time costs of mutation testing.

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

The analysis of the subsumption relationships also helped us to identify opportunities for additional optimizations. As it can be seen from Table 13, the EHC, FVR, ILR, LSC, UORD and VVR operators, which do not foresee an optimized version, generated several subsumed mutants on some projects. In Table 14 we show the specific couples of subsuming and subsumed replacement rules, where the mutants generated by the latter rule (Col. 3) were always subsumed by the mutants generated by the former rule (Col. 2). Specifically, we report those rules that held for most projects (Col. 4). As it can be seen, some replacement rules of the FVR and VVR operators consistently generated subsumed mutants for 3 or 4 out of 5 projects. Clearly, these subsumption relationships do not have universal validity. However, they are more likely to hold in other projects, and therefore might be good candidates for operator optimizations.

Lastly, Table 15 shows the time costs for running both the experiments. For each DApp, the table shows the generation time (expressed in seconds) for the optimized and the non-optimized set of mutants, respectively. This represents the time needed to generate the mutations for all target contracts. As it can be observed the generation of the mutants is rather effective and is not significantly affected by the operator optimizations. The table also shows the time (expressed in minutes) needed to perform an entire mutation testing run on each project. This value is affected by several characteristics of the project under test, such as the complexity of the test suite and the number of mutation opportunities present in each contract. External parameters, such as the hardware specifications of the workstation, can impact the mutation testing time as well. As it can be observed the time to run the assessment of the test suite can be rather high; in the case of Non-Optimized operators, Alice required around 12 hours to be tested. However, testing the same project using the simplified mutation rules allowed us to save more than 3 hours. On average, enabling operator optimizations permits to decrease the test time by $\sim$ 30%.

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

Table 16
Bionic Event DApp metrics.

Mutated Contracts	LOC	Test Suite size	Test LOC	Stmt Cov.	Branch Cov.
Event.sol EventFactory.sol	182	25	261	90	65

Answer to RQ2: Using the Optimized operators resulted in almost identical Mutation Scores as the Non-Optimized operators while limiting the generation of subsumed mutants and decreasing the time cost by $\sim$ 30% on average. Nevertheless, the Non-Optimized operators generated various unique mutants that survived testing and could represent the best choice when DApp reliability is essential.

6.4. Improving a test suite using SuMo

To answer RQ3 we selected a case study, and we enhanced its test suite based on its survivors. In particular, We selected the **Bionic Event**[21] project, as it offers a good balance between test suite complexity (see Table 16) and number of generated mutants. Bionic Event is a DApp that supports the creation and management of events, as well as the purchase, validation, and transfer of tickets. In order to enhance its test suite, we: (1) *performed a full mutation testing run* on the application and we (2) *manually analyzed the live mutants*. In particular, for this assessment, we enabled the operator optimizations as they reduce the manual effort needed to discard equivalent mutants. After (3) *designing new or improved test data* we (4) *repeated the mutation testing run*. Table 17 shows the results of the experiment before and after the enhancement of the original test suite. The second mutation testing run resulted in a significantly higher Mutation Score (Col. 6), which suggests the higher fault-detection capability of the enhanced test suite.

Indeed, examining the survivors allowed us to improve the existing test data and craft additional test cases. Several mutation operators have proved to be particularly useful during this experiment. The *Binary Operator Replacement (BOR)* operator exposed issues concerning the testing of boundary values, as well as the presence of an unused function modifier in the contract. The *Event Emission Deletion (EED)* operator generated 4 survivors; these mutants were not killed because the test suite never checked for the

[21] https://github.com/Mikearaya/bionic_event_DApp

Table 17
Bionic Event DApp results before and after enhancing the test suite.

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

Legend: *M* = total mutants; *S* = stillborn mutants, *D* = equivalent and redundant mutants discarded by the TCE, *T* = mutants under test, *MS* = Mutation Score.

correct emission of events. One of these mutants also allowed us to spot a bug in the contract. The `collectPayment()` function (listing 16) emits an event right after a selfdestruct instruction. However, once the contract is destroyed further instructions will not be executed. This prevents the event from ever being fired. Thus, we moved the event emission prior to the selfdestruct instruction and we derived a test capable of killing the resulting mutant. As shown in Listing 17, the test issues a transaction to purchase a ticket from the `firstEvent` contract on behalf of `accounts[1]`, a generic buyer (line 2). Once this asynchronous operation is completed, the test invokes the `collectPayment()` function (line 3); this transaction is issued from the current account, which is the owner of the `firstEvent` contract. When the outcome of the second transaction is available, the test checks the events emitted during the transaction (line 5). Here, we used the `eventEmitted` function offered by the `truffle-assertions` library[22] to check that an event with type `PaymentCollected` has been emitted by the transaction with result `tx`. If the event is not found (or if the event parameters are not as expected), the test fails. This test would have easily detected the bug present in the original contract.

Listing 16: EED mutant from the Bionic Event DApp

```
1  //Original
2  function collectPayment() onlyOwner public {
3     selfdestruct(msg.sender);
4     emit PaymentCollected(address(this), msg.sender,
          address(this).balance);
5  }
6
7  //Mutant
8  function collectPayment() onlyOwner public {
9     selfdestruct(msg.sender);
10    //emit PaymentCollected(address(this), msg.sender,
          address(this).balance);
11 }
```

Listing 17: Killer of the EED mutant

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

22 Truffle-assertions: https://github.com/rkalis/truffle-assertions.

Some mutants generated by the *Exception Handling Change (EHC)* operator survived due to the presence of faulty test cases. Listing 18 shows one of the live EHC mutants, while listing 19 shows the test case that was supposed to kill it. This test generates a transaction that purchases more than the maximum allowed amount of tickets for an event. The transaction is supposed to revert when the first require statement of the `purchaseTicket` function (see listing 18) evaluates the number of requested tickets. Our EHC operator commented out the require statement in question, which should cause the invocation to `purchaseTicket` to succeed (and the test to fail). The survival of this mutant indicates that the test still observed a revert despite our mutation. The problem is that the test transaction does not send enough funds for purchasing 6 tickets; this unrelated revert – originated by the check on line 5 – makes the test pass although an entire require statement was removed from the contract. Thus, potential faults in the implementation of the first require statement would go unnoticed. To fix this test and assess the correct behavior of the `purchaseTicket` function we increased the amount of wei sent by the test transaction (line 5 of listing 19). Additional tests were designed to ensure that, for instance, a user cannot buy more than the available amount of tickets left for the event.

Listing 18: EHC mutant from the Bionic Event DApp

```
1  //Original
2  function purchaseTicket (uint quantity ) public payable {
3     require(quantity <= MAX_PURCHASE);
4     require(available  >= quantity);
5     require(msg.value >= ticketPrice.mul(quantity));
6     ...
7  }
8
9  //Mutant
10 function purchaseTicket (uint quantity ) public payable {
11   //require(quantity <= MAX_PURCHASE);
12   require(available  >= quantity);
13   require(msg.value >= ticketPrice.mul(quantity));
14    ...
15 }
```

Listing 19: Faulty test case

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

The *Modifier Deletion (MOD)* operator was also useful for enhancing the test suite. The live mutant in listing 20 indicates that the false branch of the `onlyOwner` modifier is never exercised by the existing tests. Listing 21 shows the test we implemented for ensuring that a buyer cannot validate his/her own ticket.

Listing 20: MOD mutant from the Bionic Event DApp

```
//Original
function isTicketValid(address _owner,
uint _tokenId) onlyOwner  external {
    if(ownerOf(_tokenId) == _owner) {
        _burn(_tokenId);
    }
}

//Mutant
function isTicketValid(address _owner,
uint _tokenId) external {
    if(ownerOf(_tokenId) == _owner) {
        _burn(_tokenId);
    }
}
```

Listing 21: Killer of the MOD mutant

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

The transaction issued on line 3 purchases a ticket from the contract `firstEvent` on behalf of `accounts[1]` (a standard non-owner account). The second transaction on line 5 retrieves the array of tickets owned by `accounts[1]`, which contains the previously purchased ticket. Finally, on line 7 the test invokes the `isTicketValid` function from `accounts[1]`, passing the purchased ticked as a parameter. Since only the owner of the contract is allowed to perform this operation, the test awaits for a transaction revert. Thus, the MOD mutant in listing 20 can be easily killed by this test.

Answer to RQ3: Using SuMo on a real-world project allowed us to significantly increase the Mutation Score achieved by its test suite. The live mutants helped us to identify faulty test cases as well as a bug in the smart contract code.

7. Threats to validity

In this section we present the threats to the validity of our study, distinguishing four different aspects: construct validity, internal validity, external validity, and reliability (Runeson and Höst, 2009). Here we address such threats and explain the measures we employed to reduce the risk of bias.

7.1. Threats to construct validity

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

7.2. Threats to internal validity

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

Another threat lies in the correctness of the mutation process itself. To generate the mutants we parse the source code of each smart contract with the `solidity-parser-antlr` module, and then we apply changes at the Abstract Syntax Tree (AST) level. To ensure that each mutation operator behaves correctly, we manually checked the mutants to ensure that they contain the expected mutation.

Lastly, the mutants are deployed and tested on a local Ganache blockchain. When a Ganache instance is started, it sets up several accounts with a certain balance for performing transactions. Although SuMo relies on the network configuration specified by the developers, deploying and testing multiple mutants on the same Ganache instance can drain the balance of the accounts and affect the test outcomes. Indeed, the balance might not be automatically restored after an account completes a transaction. Thus, if all the mutants are tested on the same blockchain instance, some of them might be killed due to a lack of funds, rather than an actual test failure. To solve this issue without altering the initial balance configuration, SuMo spawns a new Ganache instance whenever a mutant must be deployed and tested.

7.3. Threats to external validity

The threats to external validity concern the generalizability of our findings. To validate SuMo we run mutation testing on a small set of open-source Solidity applications. Thus, our findings might not apply to other projects that offer different mutation opportunities. To mitigate this risk, we tried our best to select heterogeneous applications belonging to different domains, and with test suites of different complexities. Although we choose 5 distributed applications, some of which (e.g., Alice and Safe-Contracts) feature a relatively big number of smart contracts and test cases, we recognize that they are insufficient to establish universal validity for the given results.

On the same note, in order to practically assess the usefulness of the mutation operators, we used SuMo for improving the test suite of a single project (i.e., Bionic Event). Improving a test suite is a time-consuming activity that also requires extensive knowledge of the application under test. Analyzing the mutants helped us to improve the test suite and to spot a not previously reported bug on the project. Nevertheless, to derive a definitive answer SuMo must be adopted to improve more complex test suites, and on a larger set of projects where it is possible to apply all the mutation operators.

7.4. Threats to reliability

This category describes the extent to which the data is dependent on the conducted research, and whether the results can be reliably reproduced. The main threat to the reliability of our findings concerns the presence of flaky tests in the test suites shipped with the selected projects. These are tests that can exhibit pass and fail outcomes on different re-runs, although neither the code under test nor the test have changed. As shown by Shi et al. (2019), the Mutation Score achieved by a project might not be the same on two identical re-runs if such tests are present. For instance, a mutation m might be killed by a test t during a run $r1$. If the same mutant m is not covered by the same test t during the next run $r2$, it will be marked as live. To mitigate the risk of unreliable Mutation Scores, we re-run each test suite on the original project several times to weed out applications with possible non-deterministic tests. Even so, we cannot guarantee the absence of such tests in the selected projects, as some flaky test behavior might be difficult to observe after a few re-runs, and might also be dependent on the injected mutation. Lastly, reliability might be affected by undetected bugs in the automated tools employed in this study. To diminish this concern, we used well-supported tools, like Ganache, to conduct the mutation testing campaigns. We also mitigate the risk of faults in SuMo by (manually) testing our implementation on a few projects to ensure that everything works as expected.

8. Related work

Mutation testing. The literature on mutation testing has contributed a large set of approaches and tools for improving the effectiveness and usability of this technique. Jia and Harman (2011) provide a comprehensive overview of the available cost reduction research work, while Papadakis et al. (2019) present a survey of recent advances in mutation testing, exploring the main challenges to be tackled for the future development of the method. Kintis et al. (2010) investigate techniques for reducing mutation testing expenses while maintaining its effectiveness, such as higher-order and weak mutation testing. Other works propose approaches for dealing with stubborn mutants that are hard to kill, as they can decrease the performance of the mutation testing approach. Yao et al. (2014) provide evidence of the uneven distribution of stubborn mutants, which could be tackled through appropriate operator prioritization techniques. Dang et al. (2020) seek stubborn mutants and generate test data to kill them, while Chekam et al. (2021) propose a Dynamic Symbolic Execution technique. Several works focus on the generation of test data based on mutation testing. Yao et al. (2020) strengthen the defect-detection capabilities of the test suite through a test data generation approach for weak mutation testing. Dang et al. (2021) propose FUZGENMUT, a framework that combines machine learning and data generation techniques. Clustering the mutants and generating test data for different clusters can enhance the mutant-killing capabilities while reducing the cost of mutation testing. Lastly, Shi et al. (2019) tackle the issue of flaky tests, whose non-deterministic outcomes can compromise the reliability of the mutation testing approach.

Mutation testing for ethereum smart contracts. In the literature, it is possible to find some alternative proposals that implement a mutation testing approach for Ethereum smart contracts. Wu et al. propose **MuSC** (Wu et al., 2019; Li et al., 2019), the first mutation testing framework for the assessment of Ethereum smart contract test suites. The tool is open-source and can be found on Github. This work proposes nine JavaScript-oriented operators and fifteen Solidity-specific mutation operators that can inject faults in the smart contract code. The proposed set of operators is moderately compact, which allows limiting the total number of generated mutants. However, it lacks several operators that can introduce severe faults or vulnerabilities into the contract, such as the ones that target function modifiers. While the mutation operators used by MuSC were designed starting from the Solidity documentation, the approach proposed by Andesta et al. (2020) is based on the study of known bugs in Solidity smart contracts. This set of Solidity operators was included in **Universal Mutator** (Groce et al., 2018), a popular regexp-based mutation testing tool that can easily be adapted to different programming languages. The core idea of Andesta et al.'s work is that designing operators capable of recreating real-world bugs can be an effective way of selecting good test cases. This led to the definition of 10 classes of operators that cover many types of smart contract vulnerabilities. The results show that the proposed set leads to the generation of a big amount of invalid and redundant mutants, while the rate of generation of equivalent mutants was not mentioned. The evaluation carried out by Andesta et al. shows that the proposed operators can successfully recreate 10 out of 15 buggy contracts. However, no actual Mutation Scores were provided by the authors. Chapman et al. (2019) proposes **Deviant**, an open-source mutation testing tool that aims to help developers in delivering higher quality code. In this case, a broader set of 62 Solidity-specific mutation operators was designed according to the Solidity fault model. However, it ends up generating a large number of mutants that require a big computational effort to be executed against the test suite. The **ContractMut** tool proposed by Hartel and Schumi (2020) implements general mutation operators derived from the minimum standard Mothra set, as well as four Solidity-specific mutation operators. In order to define a more manageable set of operators and limit the number of generated mutants, the authors apply *selective mutation* techniques. In particular, the mutation operators were designed to only target the most severe vulnerabilities, with a focus on mistakes concerning access control. As a result, the percentage of stillborn mutants generated by this framework is rather small (15.8%). The effectiveness of this approach was evaluated by producing replay tests derived from historic transaction data on the blockchain. Replay tests have a limited code coverage with respect to a real test suite, which would likely achieve higher Mutation Scores.

Smart contracts flaws and vulnerabilities. Several works attempt to analyze or classify bugs and vulnerabilities that can be found in Ethereum smart contracts. Atzei et al. (2017) provide a comprehensive taxonomy of common programming flaws which may lead to actual vulnerabilities. The authors distinguish the vulnerabilities according to the level where they are introduced (Solidity, EVM bytecode, or blockchain), and they provide examples of real exploit patterns. Dika and Nowostawski (2018) thoroughly review known vulnerabilities, as well as the security code analysis tools used to identify them. Groce et al. (2020) provide a summary of flaws detected in paid security audits performed at a leading company in blockchain security. Such analysis helps to understand the likelihood and severity of real smart contract flaws, many of which can be currently reproduced by SuMo. Indeed, the proposed mutation operators can simulate bugs and/or vulnerabilities of the following classes: Access Control, Numerics, Undefined Behavior, Authentication, Reentrancy, Error Reporting, Timing, Coding-Bug, Auditing and Logging, Missing Logic, and Cryptography. For instance, the mutation operators that target function modifiers can help to simulate those cases where access control is either too permissive or too restrictive. This is extremely relevant as access control issues seem to be amongst the highest severity bugs in smart contract development (Groce et al., 2020). Although some of the issues reported in the aforementioned works do not represent a good fit for a mutation testing approach, further studies could help us to

improve the proposed operators. For instance, the Patching class identified by Groce et al. (2020) deals with flaws that can be introduced while patching or upgrading the Smart Contract code. As reported by the authors, there seem to exist some common patterns of bad upgrade logic. These represent an ideal candidate for the definition of new mutations, especially in the context of an incremental mutation testing process.

9. Conclusions and future work

This work presented a novel mutation testing approach and a corresponding fully functional tool for Solidity smart contracts. SuMo implements a comprehensive set of mutation operators for simulating a wide range of traditional and Solidity-specific faults and vulnerabilities. The analysis of the Solidity documentation and existing tools allowed us to design 11 novel mutation operators, 7 of which target the unique features of Solidity. In particular, SuMo introduces mutation operators for the overloading mechanism, which was never considered by related works, as well as new operators for mutating the contract constructor, function modifiers, cryptographic global functions, the SafeMath library, global blockchain variables, enums, return values and explicit conversions. Besides applying the Trivial Compiler Equivalence (TCE) for reducing the impact of mutant equivalencies, SuMo also offers the possibility to enable operator optimizations so as to reduce the time and cost required for performing a mutation testing run.

We evaluated the effectiveness of SuMo by running the mutation testing process on five real-world complex DApps. The experimental results show that the test suites shipped with the selected applications cannot achieve high Mutation Scores, especially for Solidity-specific mutations. This further shows that only using a structural coverage criterion as a quality gate for the deployment of a contract can be risky, as it could not assure high reliability. Even though the selected projects did not offer opportunities for applying all the operators included in SuMo, we could test and evaluate the most relevant ones; among the novel operators, ER (Enum Replacement), GVR (Global Variable Change), SFR (SafeMath Function Replacement) and OLFD (Overloaded Method Deletion) achieved the lowest Mutation Scores. Although the Mutation Score is not an absolute indication of the operator effectiveness, the analysis of the live mutants suggests these operators can guide the design of a more effective test suite. The MCR (Mathematic and Cryptographic function Replacement), and CCD (Contract Constructor Deletion) mutants were more frequently killed by the tests, but the survivors can help to pinpoint weak test cases and poorly tested areas of code. The results suggest that the Non-Optimized operators can provide higher reliability during a full-scale mutation analysis, while the simplified mutations can significantly speed up the mutation testing process. This mechanism allows smart contract developers to select the operators that best fit their needs. To further assess the effectiveness of the operators, we enhanced the test suite of a selected case study based on its live mutants. The analysis allowed us to identify faulty test cases and smart contract code, and to increase the Mutation Score of the provided test suite.

Future research directions include evaluating the effectiveness of the mutation operators that were not applied during our experiments, and implementing optimized versions of those operators that consistently generated subsumed mutants on several projects. On the same note, running SuMo on more projects can help to identify which subsumption relationships are more frequently established among the implemented mutation rules, and drive the enhancements of the operator optimizations. Indeed, knowing which subsumption rules have a general validity can help to reduce the number of subsumed mutants while minimizing the risk of excluding useful mutations. Conducting a more extensive study of smart contract flaws taxonomies and ensuring the injection of up-to-date, meaningful mutations is also essential to guarantee the usefulness of the proposed approach. It is worth mentioning that, in order to be economically acceptable, SuMo has to include mechanisms for incremental mutation testing. Indeed, as it has been observed for the Alice project, generating and testing an extensive set of mutants can require a significant amount of time and computing resources. Therefore, the testers must wait for a long time before they receive useful feedback. While running a stand-alone mutation testing campaign is still reasonable, repeating the entire process each time the DApp evolves can be undesirable. To make mutation testing more affordable, especially on very large codebases, we envision the usage of SuMo in an incremental process, where after each commit an assessment of the test suites is performed (Ma et al., 2020). Thus, future work should focus on expanding SuMo with an automatic regression mutation testing feature capable of: (1) generating mutants for the areas of code that were affected by some changes since the previous program revision, and (2) applying regression techniques for selecting test cases that exercise the evolved parts of the contract. Such improvements can save time and computing resources, making SuMo more economically acceptable in real development settings (Zhang et al., 2012). Understanding whether certain mutation types are especially effective during incremental mutation testing is also an interesting research direction. For instance, mutations capable of reproducing bad upgrade logic patterns (Groce et al., 2020) might be a suitable candidate. Clearly, further investigations are needed to identify the nature of such patterns and to understand whether they overlap with some existing operators. Lastly, integrating the Equivalent Mutant Detector with SuMo can remove the manual step required to the tester and improve the usability of the tool.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Morena Barboni's work has been partially supported by the Italian MIUR PRIN 2017 Project: SISMA (Contract 201752ENYB).

References

Andesta, E., Faghih, F., Fooladgar, M., 2020. Testing smart contracts gets smarter. In: 2020 10th International Conference on Computer and Knowledge Engineering (ICCKE). IEEE, pp. 405–412.

Atzei, N., Bartoletti, M., Cimoli, T., 2017. A survey of attacks on ethereum smart contracts (sok). In: International Conference on Principles of Security and Trust. 10204, Springer, pp. 164–186.

Barboni, M., Morichetta, A., Polini, A., 2021. Sumo: A mutation strategy for solidity smart contracts. In: 2021 IEEE/ACM International Conference on Automation of Software Test (AST). ACM, pp. 50–59.

Buterin, V., 2013. Ethereum whitepaper. https://ethereum.org/whitepaper/.

Chapman, P., Xu, D., Deng, L., Xiong, Y., 2019. Deviant: A mutation testing tool for solidity smart contracts. In: International Conference on Blockchain. IEEE, pp. 319–324.

Chekam, T.T., Papadakis, M., Cordy, M., Traon, Y.L., 2021. Killing stubborn mutants with symbolic execution. ACM Trans. Softw. Eng. Methodol. 30 (2), 19:1–19:23.

Corradini, F., Marcelletti, A., Morichetta, A., Polini, A., Re, B., Tiezzi, F., 2020. Engineering trustable choreography-based systems using blockchain. In: Symposium on Applied Computing. In: SAC '20, ACM, pp. 1470–1479.

Corradini, F., Marcelletti, A., Morichetta, A., Polini, A., Re, B., Tiezzi, F., 2022. Engineering trustable and auditable choreography-based systems using blockchain. ACM Trans. Manage. Inf. Syst. 13 (3).

Dang, X., Gong, D., Yao, X., Tian, T., Liu, H., 2021. Enhancement of mutation testing via fuzzy clustering and multi-population genetic algorithm. IEEE Trans. Softw. Eng..

Dang, X., Yao, X., Gong, D., Tian, T., 2020. Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain. 69, pp. 334–348.

Deng, L., Offutt, J., Li, N., 2013. Empirical evaluation of the statement deletion mutation operator. In: International Conference on Software Testing, Verification and Validation. IEEE, pp. 84–93.

Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., Hierons, R., 2018. Smart contracts vulnerabilities: a call for blockchain software engineering? In: International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, pp. 19–25.

Dika, A., Nowostawski, M., 2018. Security vulnerabilities in ethereum smart contracts. In: Internet of Things and Green Computing and Communications and Cyber, Physical and Social Computing and Smart Data. IEEE, pp. 955–962.

Frankl, P.G., Weiss, S.N., Hu, C., 1997. All-uses vs mutation testing: An experimental comparison of effectiveness. J. Syst. Softw. 38 (3), 235–253.

Groce, A., Feist, J., Grieco, G., Colburn, M., 2020. What are the actual flaws in important smart contracts (and how can we find them)? In: Financial Cryptography and Data Security. Springer International Publishing, pp. 634–653.

Groce, A., Holmes, J., Marinov, D., Shi, A., Zhang, L., 2018. An extensible, regular-expression-based tool for multi-language mutant generation. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. ACM.

Hartel, P.H., Schumi, R., 2020. Mutation testing of smart contracts at scale. In: International Conference TAP@STAF 2020. In: LNCS, 12165, Springer, pp. 23–42.

Honig, J.J., Everts, M.H., Huisman, M., 2019. Practical mutation testing for smart contracts. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology. Springer, pp. 289–303.

Inozemtseva, L., Holmes, R., 2014. Coverage is not strongly correlated with test suite effectiveness. In: 36th International Conference on Software Engineering - ICSE. ACM, pp. 435–445.

Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.

Kintis, M., Papadakis, M., Malevris, N., 2010. Evaluating mutation testing alternatives: A collateral experiment. In: 2010 Asia Pacific Software Engineering Conference. IEEE, pp. 300–309.

Kurtz, B., Ammann, P., Delamaro, M.E., Offutt, J., Deng, L., 2014. Mutant subsumption graphs. In: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, March 31 - April 4, 2014, Cleveland, Ohio, USA. IEEE Computer Society, pp. 176–185.

Li, Z., Wu, H., Xu, J., Wang, X., Zhang, L., Chen, Z., 2019. MuSC: A tool for mutation testing of ethereum smart contract. In: International Conference on Automated Software Engineering, ASE 2019. IEEE, pp. 1198–1201.

Ma, Y.-S., Jeff, O., Rae, K.Y., 2005. MuJava: an automated class mutation system. Softw. Test. Verif. Reliab..

Ma, W., Laurent, T., Ojdanić, M., Chekam, T.T., Ventresque, A., Papadakis, M., 2020. Commit-aware mutation testing. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 394–405.

Ma, Y.-S., Offutt, J., Description of class mutation mutation operators for java. https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf.

Ma, Y.-S., Offutt, J., Description of muJava's method-level mutation operators. https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf.

Mehar, M.I., Shier, C.L., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., Kim, H.M., Laskowski, M., 2019. Understanding a revolutionary and flawed grand experiment in blockchain. J. Cases Inf. Technol. 21 (1), 19–32.

Miraz, M.H., Ali, M., 2020. Blockchain enabled smart contract based applications: Deficiencies with the software development life cycle models. Inf. Syst. eJ..

Papadakis, M., Jia, Y., Harman, M., Traon, Y.L., 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M., 2019. Mutation Testing Advances: An Analysis and Survey. Elsevier, pp. 275–378.

Porru, S., Pinna, A., Marchesi, M., Tonelli, R., 2017. Blockchain-oriented software engineering: challenges and new directions. In: 39th International Conference on Software Engineering Companion (ICSE-C). IEEE, pp. 169–171.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14 (2), 131–164.

Shi, A., Bell, J., Marinov, D., 2019. Mitigating the effects of flaky tests on mutation testing. In: ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 296–306.

Tengeri, D., Vidacs, L., Beszedes, A., Jasz, J., Balogh, G., Vancsics, B., Gyimothy, T., 2016. Relating code coverage, mutation score and test suite reducibility to defect density. In: Int. Conf. on Software Testing, Verification and Validation (ICSTW). IEEE, pp. 174–179.

Wood, G., 2021. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf.

Wu, H., Wang, X., Xu, J., Zou, W., Zhang, L., Chen, Z., 2019. Mutation testing for ethereum smart contract. ArXiv abs/1908.03707.

Yao, X., Harman, M., Jia, Y., 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Proceedings of the 36th International Conference on Software Engineering. In: ICSE 2014, ACM, pp. 919–930.

Yao, X., Zhang, G., Pan, F., Gong, D., Wei, C., 2020. Orderly generation of test data via sorting mutant branches based on their dominance degrees for weak mutation testing. IEEE Trans. Softw. Eng..

Zhang, L., Hou, S.-S., Hu, J.-J., Xie, T., Mei, H., 2010. Is operator-based mutant selection superior to random mutant selection? In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 1. ACM Press, pp. 435–444.

Zhang, L., Marinov, D., Zhang, L., Khurshid, S., 2012. Regression mutation testing. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012. ACM Press.

Zheng, Z., Xie, S., Dai, H., Chen, X., Wang, H., 2017. An overview of blockchain technology: Architecture, consensus, and future trends. In: International Congress on Big Data. IEEE, pp. 557–564.

Zou, W., Lo, D., Kochhar, P.S., Le, X.-B.D., Xia, X., Feng, Y., Chen, Z., Xu, B., 2021. Smart contract development: Challenges and opportunities. IEEE Trans. Softw. Eng. 47, 2084–2106.