Mutation Testing for Smart Contract Inspection

***Abstract*—Smart contracts hold the potential for revolutionizing various industries, but their implementation requires thorough testing due to the associated financial risks. Mutation testing is a powerful technique that can boost the fault-detection capabilities of a test suite, but it can also foster a deeper understanding of the smart contract behavior. This work investigates the productivity of mutants with respect to their capabilities in disclosing Solidity issues. Based on these findings, it proposes an enhanced mutation strategy to better assist smart contract auditors during code inspection activities. 9 novel mutation operators are proposed in this paper and 13 existing operators are improved. The results show a 30% reduction in the number of generated mutants and time savings of 62%, while increasing the set of productive mutants related to issues by 43% overall. We note that the most valuable type of mutants that could help disclose an issue as a result of manual mutant inspection was increased by 125%.**

***Index Terms*—Mutation Testing, Smart Contract, Auditing**

I. Introduction

Smart contracts have the potential to revolutionize various industries by automating transactions and eliminating the need for intermediaries. However, the risks associated with their implementation require in-depth testing before real-world deployment [1], [2]. With the growing recognition of such risks, specialized *auditing* companies started offering services to review the security and functionality of smart contracts. Through their expertise, auditors conduct rigorous code inspections, execute automatic tools (e.g., Slither [3] and Echidna [4]), and evaluate the quality of the provided test suite. Although coverage analysis helps to identify under-tested areas of code, it cannot provide in-depth insights about the codebase [5]. Indeed, the effects of covered statements might never be asserted by the tests [6]. *Mutation testing*, long considered the most powerful test adequacy assessment technique [7], is a promising approach to support auditors in their activities. This technique evaluates the capabilities of a test suite in detecting program mutations that mimic real-world bugs [8]. Solidity mutants have been recognized for their potential in improving smart contract test suites [9], [10], yet their relationship with code inspection remains largely unexplored. Mutation analysis could offer a unique perspective to auditors, helping them to reason about the code and fostering a deeper understanding of the contract's behavior. To investigate this aspect we run SuMo[1], an open-source Solidity mutation testing tool [11], on a set of projects audited at the partner blockchain security company. In prior work, Barboni et al. provided a comprehensive description of the operators implemented by SuMo [9]. Building upon such work, we assess the value of live mutants in the code review process. By evaluating the implications of a mutant's survival, it may be possible to uncover implicit assumptions, design flaws, or even bugs in the contract under test. Insights gained from auditors can be invaluable in further refining the operators and incorporating mutation testing in the code review workflow.

[1]SuMo: https://github.com/MorenaBarboni/SuMo-SOlidity-MUtator

II. Background

Mutation testing ensures the adequacy of test data in detecting real faults by introducing small defects into the codebase. Mutants have demonstrated their effectiveness as test goals, leading to increased adoption in industrial settings [12], [13]. Notably, the effectiveness of this technique is reliant on the selection of mutation operators [14]. Available mutation testing tools for Solidity [9], [10], [15] aim to inject meaningful modifications that result in a semantically different program (i.e., a *non-equivalent* mutant). This aligns with the expectation that a defect should be detectable through testing. However, some mutants, though seemingly non-killable, can still be marked as productive [16]. These mutants are often addressed through subsequent code patching or refactoring, making them valuable during code review.

III. Methodology

In the following, we will outline the research questions we aim to answer in this work and the relative methodologies.

A. RQ1 - How can mutation analysis support smart contract auditing activities?

RQ1 investigates how mutants generated by SuMo can be actionable to smart contract auditors. To assess their productivity, we will perform mutation analysis on a set of projects (see Section III-C) that were submitted to the partner company for an audit. Our evaluation involves examining whether the subject contracts contain **live mutants** that can be **linked to issues** disclosed by auditors in the past. Our definition of productive mutant builds upon Petrovic et al.'s work [16], extending the concept to encompass the role of mutants in issue discovery. Specifically, we classify a mutant as **productive** if it either: a) facilitates the derivation of a test that might reveal an issue, or b) directly exposes the presence of an issue during code review. Then, for every issue found in each audited project, we will perform the following steps:

1) Analyze the characteristics of the finding, how it impacts the program behavior, and the recommended fix;
2) Assess whether one or more live mutants produced by SuMo could have helped to disclose the issue.
3) Tag the issue based on whether it was associated to:

a) **No Productive Mutants (NPM)**: no live mutants could have helped to disclose the issue;
b) **Test-Productive Mutants (TPM)**: one or more mutants could have helped to disclose the issue as a result of test improvement;
c) **Inspection-Productive Mutants (IPM)**: one or more mutants could have helped to disclose the issue as a result of direct mutant inspection;

Note that an issue might be associated with both Test and Inspection productive mutants. Of particular interest to us is the latter category, as it can highlight mutation operators with the potential to assist auditors in their code inspection efforts.

B. RQ2 - How can mutation operators be enhanced to better support smart contract auditors?

RQ2 aims to understand how mutation testing can be enhanced to better support auditing activities. Besides analyzing the productive mutants generated by SuMo, valuable feedback from auditors will be collected to evaluate the effectiveness and practicality of the existing operators. The resulting improvements will be evaluated following the same procedure described for RQ1. Findings from this study will pave the way for the integration of mutation testing into the smart contract auditing workflow. Indeed, the identification of the most valuable mutations is essential for reducing the cognitive overhead placed on auditors during mutation analysis.

C. Experimental Subjects

The study was conducted on 8 Solidity projects that were submitted for a security assessment to our industrial partner. To ensure a meaningful analysis, we only selected projects accompanied by a test suite with good test coverage. For each project, we meticulously reviewed the associated security report and excluded those issues that cannot reasonably be addressed through mutation testing. Examples of such findings include documentation improvements or the incorporation of new logic due to specification changes. These lay beyond the scope of mutation testing, which primarily helps to ensure the correctness of existing code logic. In Table I we report the details about the projects provided by the company. Due to the sensitive nature of the data, we identify each subject with a unique identifier (ID). We also report the number of contract files (#CF), contract lines of code (#LOC), and test methods (#TM) in the codebase with the total statement coverage (note that project 3 displays a low value, but most relevant contracts for the audit were thoroughly covered by the tests). Lastly, we report the total number of issues disclosed by auditors, and those that are relevant for the analysis with mutation testing (Mutation Testing Relevant Issues, or #MTRI), grouped by severity level (*Low*, *Medium*, *High* or *Other*).

TABLE I: Experimental Subjects

ID	#LOC	#CF	#TM	Stmt. Cov.	Total Issues	#Low MTRI	#Medium MTRI	#High MTRI	#Other MTRI
1	2105	14	83	96.9	21	2	4	-	-
2	564	11	141	90	29	4	4	2	2
3	3102	42	155	57.4	25	3	4	2	-
4	1815	19	96	98	48	9	5	6	2
5	1678	10	118	91.4	26	4	1	3	-
6	641	10	165	98.8	11	3	3	-	-
7	68	1	11	82.6	7	3	-	-	-
8	2745	21	339	75.6	18	8	-	1	4
Tot.	12718	128	1108	-	185	36	21	14	8

Legend: LOC = lines of code, CF = contract files, TM = test methods, $Stmt.Cov.$ = statement coverage $MTRI$ = mutation testing relevant issues

TABLE II: Results for the original SuMo operators

ID	$\#G$	$\#V$	MS	Total MTRI	MTRI with NPM	MTRI with TPM	MTRI with IPM	MTRI with TPM &IPM
1	3177	2720	55.6	6	4	2	0	0
2	727	622	63.8	12	8	4	0	0
3	3836	3077	64.3	9	7	2	0	0
4	2069	1791	51	22	11	8	1	2
5	2743	2320	69.3	8	3	2	2	1
6	922	692	81.5	6	3	2	1	0
7	134	117	69.2	3	2	1	0	0
8	5171	4589	43.3	13	4	4	1	4
Tot.	18779	15928	-	79	42	25	5	7

IV. Answering RQ1

Table II shows the mutation testing results obtained using the original set of SuMo operators. For each project (ID), we report the total number of *generated mutants* ($\#G$), *valid mutants* ($\#V$) that are not *uncompilable*, *equivalent* or *timedout*, and the achieved Mutation Score (MS). Overall, SuMo generated 18779 mutations which were tested in about 302 hours. Although most audited applications were submitted with high-quality test suites, almost all of them achieved Mutation Scores below ideal levels. This suggests that the test cases may not be as thorough as expected and issues might be present within the contract code. For each project, Table II also shows the number of Mutation Testing Relevant Issues (MTRI) related to No Productive (NPM), Test-Productive (TPM) and Inspection-Productive (IPM) mutants (as defined in Section III-A). The original mutation operator set generated productive mutants for ~47% of the issues reported by auditors. Although most of them (32%) are related to TP mutants, a small percentage of issues (~15%) feature mutants that could have provided useful insights during code inspection. An example of such mutant is shown in Listing 1. The `_getData` function includes a low-severity *Incorrect Validation* on data retrieved from an external oracle. Indeed, it relies on a price input that may assume a zero value, a case that the smart contract does not account for. However, the relational mutation injected by SuMo is essentially equivalent to the patch that clients applied after receiving the audit report. While not all IP mutants necessarily serve as patches, in this instance, the live mutant could have suggested how to refactor the contract.

Listing 1: Example of Inspection-Productive Mutant

```
function _getData(...) internal returns(...) {
++   while (latestPrice <= 0) { ... }
--   while (latestPrice < 0) { ... }
}
```

The effectiveness of mutation analysis is somewhat consistent across different issue severity levels. Productive mutants were generated at a rate of ~50% for High, ~37% for Medium, and ~47% for Low Severity MTRI. Table III further breaks down the results with respect to each category of issue, revealing distinct patterns in the generation of productive mutants. Categories such as *Miscalculation*, *Logical Oversight*, *Precision Loss*, and *Missing or Incorrect Validation* exhibited a substantial number of related TP and IP mutants. These categories often encompass issues that are more commonly encountered in smart contracts, leading to a greater number of mutation opportunities. Conversely, issue categories like *Access Control*, *Block Dependence*, *Reentrancy*, and *Unchecked Transfer* had limited or no productive mutants associated with them. These issues may prove more challenging to reveal using mutation testing, primarily due to their nuanced and complex nature. In the next Section, we will investigate how SuMo's mutation operator performed throughout the experiment to explore potential enhancements opportunities.

V. Improvement of the mutation operators

In this Section we discuss the performance of the SuMo operators on the audited projects (we recall that its operator set was comprehensively described in prior work by Barboni et al. [9]). Based on such analysis, supported by practical feedback from auditors, we further optimize the set by removing uninteresting rules, improving the existing ones, or introducing new operators altogether. For each original SuMo operator, Table IV shows the number of mutants it generated ($\#G$) and successfully tested ($\#V$) across all projects, and its Mutation Score. It also reports the number of issues for which that operator generated at least one Test-Productive (TPM) or Inspection-Productive (IPM) mutant. Operators that generated at least 1 productive mutant during the experiments are highlighted in bold.

A. *Block and Transaction Properties*

Mutations related to block and transaction properties can hold significant value for auditors. As noted in Table IV, GVR produced a TP mutant related to a time miscalculation issue, and an IP mutant with the potential to reveal a high-severity block variable dependence. The latter simulated the actions of block proposers manipulating variables to achieve a desired outcome, influencing the behavior of the smart contract. However, substituting each block variable with every compatible option proved to be an inefficient approach. This method resulted in many redundant mutations, inflating the Mutation Score and wasting valuable time.

TABLE III: Productive mutants for each category of issue

Category of Issue	Total MTRI	MTRI with NPM	MTRI with TPM	MTRI with IPM	MTRI with TPM &IPM
Access Control	7	6	0	1	0
Aliasing	1	1	0	0	0
Block Variable Dependence	1	0	0	1	0
Checks-Effects-Interactions Violation	2	1	1	0	0
Gas usage	4	0	3	0	1
Integer Over/Underflow	1	1	0	0	0
Initialization	3	2	1	0	0
Event Log Usage	1	0	0	1	0
Miscalculation	11	3	6	0	2
Logical Oversight	4	1	2	0	1
Other	3	2	1	0	0
Precision Loss	4	1	1	1	1
Reentrancy	3	3	0	0	0
Timing	1	0	1	0	0
Unchecked Transfer	5	4	0	1	0
Validation	28	17	9	0	2
Tot.	79	42	25	5	7

TABLE IV: Results grouped by Mutation Operator

MutOp Class	MutOp ID	$\#G$	$\#V$	MS	MTRI with TPM	MTRI with IPM
Overloading	ACM	2	1	0	0	0
Expressions	**AOR**	525	525	83	**3**	**0**
Types (Address)	AVR	112	99	82.8	0	0
Control	**BCRD**	44	41	63.4	**1**	**0**
Types (Standard)	**BLR**	169	169	65.7	**1**	**1**
Expressions	**BOR**	7738	7394	62.8	**19**	**3**
Control	CBD	4	4	25	**1**	**0**
Construction	CCD	30	12	75	0	0
Control	**CSC**	747	730	59	**5**	**1**
Data Location	DLR	556	64	20.3	0	0
Expressions	DOD	22	21	33.3	0	0
Conversion	ECS	76	28	60.7	0	0
Event	**EED**	223	221	25.8	**4**	**1**
Error Handling	**EHD**	655	623	47.4	**7**	**0**
Types (Enum)	ER	35	32	34.4	0	0
Ether Transfer	**ETR**	64	40	65	**0**	**1**
Visibility	FVR	1849	998	41	0	0
Block/Tx Prop.	**GVR**	500	412	59	**1**	**1**
Types (Integer)	**ILR**	1375	1303	44.9	**12**	**4**
Expressions	ICM	27	1	100	0	0
Control	**LSC**	182	167	80.2	**3**	**0**
Func. Mutability	PKD	13	3	0	0	0
Global Function	MCR	92	87	67.8	0	0
Func. Mutability	MOC	38	37	2.7	0	0
Func. Mutability	**MOD**	337	319	28.5	**5**	**0**
Func. Mutability	**MOI**	608	494	52.6	**0**	**1**
Func. Mutability	MOR	553	540	52.2	0	0
Overloading	OLFD	17	3	33.3	0	0
Overriding	ORFD	272	22	59	0	0
Return	**RSD**	470	462	68.8	**6**	**0**
Return	RVS	28	16	62.5	0	0
Libraries	SFR	52	52	5.8	0	0
Overriding	SKD	31	29	69	0	0
Overriding	SKI	85	3	33.3	0	0
Types (String)	SLR	200	130	60	0	0
Block/Tx Prop.	TOR	119	119	34.4	0	0
Expression	**UORD**	312	300	76.3	**1**	**0**
Variable Unit	**VUR**	76	68	69.1	**0**	**1**
Visibility	VVR	541	359	12	0	0

Legend: G = generated mutants, V = valid mutants, MS = mutation score.

GVR (Global Variable Replacement): The GVR operator was revised to replace block variables that are normally used as a source of randomness (e.g., `block.timestamp`), with `block.prevrandao`. The survival of this mutant likely points to a dangerous dependency where `prevrandao` should be used instead. Indeed, such a drastic mutation is more likely to be killed if the target variable is used for other purposes (e.g., to trigger time-dependent operations). Other Integer block and transaction variables, which can affect gas, Ether, and time-related computations and checks, are both incremented and decremented by one. Lastly, `blockhash(uint blockNumber)` is mutated to 0. Indeed, this function always returns 0 if the input block number is older than 256 blocks. Such scenario should be appropriately handled since this known behavior could be exploited.

TOR (Transaction Origin Replacement): TOR helps to identify risks associated with using `tx.origin` for authentication. In our experiments, all 119 mutations involved replacing `msg.sender` with `tx.origin`, as the latter is rarely used. However, the effort required to kill such mutants may outweigh the benefits. Indeed, detecting such fault requires tests that simulate transaction chains, which are impractical to define and not typically included in standard test suites. This inefficient mutation rule was therefore removed from TOR.

B. Casting

The ECS operator included in SuMo replaces each `uint` and `bytes` casting with its smallest available type (e.g., replacing `uint256` with `uint8`). However, these replacements can be too noticeable in tests because of their significant impact. SuMo also missed several mutation opportunities due to its lack of support for SafeCast, a popular OpenZeppelin's library[2] that adds overflow checks to Solidity's casting operators. Lastly, we observed that simple ECS mutations fall short in uncovering complex casting-related issues such as aliasing (see Table III).

CER - Casting Expression Replacement: The CER mutation operator was introduced to overcome the limitations of ECS. To make casting mutations more subtle, CER only decrease the type casting by one step. Additionally, CER replaces each SafeCast function invocation with a regular cast. This type of mutant can only be killed if a revert is observed upon overflow, forcing the developers to verify such a scenario. Instead, if the contract imports the SafeCast library and yet a standard casting expression is found within its code, the opposite mutation is injected. A surviving mutant likely indicates that such operations should be replaced with the corresponding SafeCast function.

AAC - Address Aliasing Check: The new AAC operator can help to discover aliasing issues by inserting multiple statements before a downcasting operation. Listing 2 shows the aliasing issue found in one of the considered subjects by auditors. Here, `_getAddressFromId(bytes32 id)` downcasts 32 bytes to an address, discarding the lower 12 bytes. As a result the same address can be returned for different ids. The new AAC operator first converts the target identifier to a dynamic byte array, then it applies the logical NOT operator to its last byte. By returning the address downcasted from the mutated identifier, it is possible to disclose the problem.

Listing 2: Example of AAC mutation

```
function _getAddressFromId(bytes32 _id) external
  ↪ returns (address) {
+    bytes memory arr = abi.encodePacked(_id);
+    arr[arr.length -1] = ~arr[arr.length -1];
+    return address(bytes20(arr));
-    return address(bytes20(_id));
}
```

[2]SafeCast: https://docs.openzeppelin.com/contracts/5.x/api/utils#SafeCast

C. Construction

CCD (Contract Construction Deletion) mutations applied by SuMo were easily detected by the tests as they remove the whole constructor. This approach can also mask more subtle problems related to individual initialization statements within the constructor. Indeed, most initialization issues from the audited projects were not related to productive mutants.

ISD (Initialization Statement Deletion): The ISD operator improves CCD by independently targeting each statement within constructors or initialize functions. As a result, it prevents single initialization operations from taking place.

D. Control Flow

Mutations involving control flow elements generated productive mutants throughout the experiments and do not necessitate major revisions. Particually, CSC showed that certain branches were not being checked by the test suite (MS = ∼56%). CSC generated IP and TP mutants for 1 and 5 issues respectively (See Table IV). These span over various issue categories, but all of them could have been detected by a more thorough test suite. Less frequent control mutations, such as those targeting loops (LSC and BCRD), generated several TP mutants as well, most of which related to inefficient implementations and gas usage issues.

E. Data Location

The Data Location Replacement (DLR) operator was not deemed useful for improving tests or disclosing issues. However, a promising research direction emerges when considering its inclusion in a distinct cluster of operators tailored for optimizing gas consumption. For example, a live `storage` to `memory` mutant suggests that storage might be unnecessary for the data in question, as state changes are not involved.

F. Data Structures

Array mutations can assist auditors in verifying the correctness of content updates. While valuable index access mutations were generated, SuMo lacked operators for targeting array manipulations through dedicated member functions.

AMR (Array Member Replacement): The AMR operator was introduced to ensure the correct state of arrays after an update operation. AMR comments out any `pop()` and `push()` invocation to simulate a missing element deletion and insertion. It also removes the argument of a `push()` to simulate the insertion of the wrong element into the array.

G. Data Types (Solidity-specific)

By altering address values, auditors can validate the effectiveness of tests in revealing various unexpected behaviors. However, our analysis of the AVR operator revealed that replacing each address value with both `address(0)` and `address(this)` often resulted in redundant mutations. In all evaluated projects, if the first replacement survived or was killed by the tests, the second replacement yielded the same outcome. Moreover, as shown by its rather high Mutation Score, the standard address faults injected by AVR can easily be detected during testing. These should be substituted with more contextually relevant mutations. Lastly, AVR overlooked several mutation opportunities associated with non-literal address assignments. For example, one high-severity access control finding from Project 4, shown in Listing 3, reveals that any non-whitelist account (line 2) can delegate voting power to a whitelisted address. In this scenario, replacing the address assigned to the delegator with a different one could have helped to expose the problem.

Listing 3: Issue related to whitelisted address from Project 4

```
1  function delegate(address delegatee, uint256 id)
       ↪ public virtual override {
2      address account = _msgSender();
3      _delegation[id][account] = delegatee;
4      ...
5  }
```

AVR (Address Value Replacement): After the results evaluation, AVR was refined to inject mutations that take into account contextual data. By preemptively collecting address identifiers AVR can recognize additional mutation opportunites. For example, the right-hand side of an assignment will be replaced with an address function parameter, with another address variable declared within the same function, or with a state variable identifier. When multiple identifiers are available, they are prioritized based on their proximity to the target so as to increase the likelihood of injecting a realistic bug. Only when no other suitable identifier is available, the target will be replaced with the standard zero address.

H. Data Types (Traditional)

The numeric (ILR) and boolean (BLR) value-targeting operators generated a substantial number of TP and IP mutants for the issues we analyzed (See Table IV), most of which were related to Miscalculation, Precision Loss and Validation. However, similar to AVR, their rules were restricted to literal values, missing out on various mutation opportunities.

NVR (Number Value Replacement), BVR (Boolean Value Replacement) and SVR (String Value Replacement): NVR, BVR and SVR are enhanced operators capable of mutating the right-hand side of simple integer, boolean and string assignments or declarations, even when the target is not a literal. This is achieved by first gathering the name and type of each function argument, local variable and state variable. Furthermore, we expanded NVR to support rational numbers and the scientific notation.

I. Error Handling

Mutations involving the deletion of error handling statements (e.g., `require`) received positive feedback from auditors. Indeed, the low Mutation Score achieved by EHD (see Table IV) demonstrates how test suites submitted for audits inconsistently explore error scenarios. While EHD only produced TP mutants during our experiments, it may also reveal interesting fix opportunities during code inspection, such as the usage of redundant error checking.

EHD (Exception Handling Deletion): The EHD operator only required a minor revision to include support for custom Solidity errors, allowing the evaluation of the test suite's effectiveness in handling user-defined error expressions.

In the context of error handling, we noted a common issue across projects: incorrect or missing validations. These generally indicate that a function is lacking proper input value checks, or that the contract state is not being verified before performing a certain action. Since mutation testing does not rely on program specification, missing validations are inherently difficult to reveal. SuMo exposed some of them through different combinations of mutants, but a dedicated operator would provide auditors with more immediate feedback.

RSI (Require Statement Insertion): RSI was designed to reason about missing validations. While identifying the location where a specific statement is missing may not be feasible, the repetitiveness of validation checks in smart contracts offers an opportunity to target common oversights. RSI analyzes all the require statements in a contract and determines whether they should be injected into other functions. Compatibility is based on the target function parameters and the existing require statements in its body. For example, the `mint` function in Listing 4 lacks a check to ensure the token recipient `to` is unfrozen. This validation is however present in the `transfer` function of the same contract. RSI can copy the `require` in the mint function, forcing the auditor to evaluate its survival.

Listing 4: Missing input validation from Project 7

```
function mint(address to,uint256 amount) public {
++  require(!_frozen[to]);
    ...
}
function transfer(address to,uint256 value) ... {
    require(!_frozen[to]);
    ...
}
```

J. Expressions

Mutations involving expressions were positively received by auditors, with few exceptions. In the following, we discuss the most relevant insights.

AOR (Assignment Replacement): Compound assignment replacements can reveal instances of values not being checked by the tests after an update, and help to disclose Miscalculation issues. However, applying multiple replacements to the same target rarely provides additional insights. The operator was refined to replace each compound assignment solely with the simple assignment, the most surviving replacement across all projects.

BOR (Binary Replacement): Relational operators are highly prevalent and linked to many TP mutants. To mitigate their testing time overhead, we pinpointed specific replacement combinations that support the analysis of positive and negative test scenarios and boost edge case coverage. Mathematical operators replacements, while not as frequent, often yielded redundant outcomes. These were similarly optimized considering the most subtle increment and decrement replacements.

ICM (Increments Mirror): The ICM operator replaces compound assignments with their mirrors (e.g., $-= \to =-$). Ultimately, ICM was removed from the set due to the limited insights it provided and its overlap with the AOR mutants.

PLR - Precision Loss Replacement: The limited precision of Solidity's fixed-point system can lead to inaccuracies when a division precedes a multiplication. As shown in Table III, simply replacing mathematical operators (BOR) or numerical values (ILR) is not the best approach to disclose precision losses. The new PLR operator swaps the order of multiplication and division in binary or tuple expressions to simulate precision loss. As shown in Listing 5, PLR also targets cases where division precedes multiplication, prompting developers to evaluate whether the original order is intentional or indicative of a potential bug.

Listing 5: Example of PLR mutant

```
function foo(uint a,uint b) public returns(uint){
-     return a / 100 * b;
+     return a * b / 100;
}
```

K. Events

Commenting out event emissions (EED) can help monitor the contract behavior by ensuring proper log verification. Indeed, the incorrect event emission reporter by auditors (See Table III) was effectively disclosed by EED. Additionally, auditors suggested the insertion of mutation operators for swapping indexed event parameters. When a Solidity event is emitted, indexed parameters are evaluated before non-indexed parameters, in right-to-left order. This unusual evaluation order might result in unexpected outcomes if the parameters have conflicting side effects[3].

EAS - Event Arguments Swap: The new EAS operator implemented by SuMo looks for function calls in event emission statements and, if they correspond to indexed parameters, it swaps them to simulate a different evaluation order.

Listing 6: Example of EAS mutation

```
event AdminFeeChanged(uint256 indexed oldFee,
    ↪ uint256 indexed newFee);

function changeAdminFees(uint256 newAdminFee)
    ↪ external onlyAdmin nonReentrant {
-   emit AdminFeeChanged(retireOldAdminFee(),
    ↪ setNewAdminFee(newAdminFee));
+   emit AdminFeeChanged(setNewAdminFee(
    ↪ newAdminFee), retireOldAdminFee());
}
```

L. Function Calls

SuMo includes several operators for mutating global function calls, regular function definitions and return statements. However, there are no operators for verifying the effects of standard function calls on the smart contract state.

FCD (Function Call Deletion): The FCD operator deletes those function calls that are not embedded into other expressions (e.g., assignments, conditional statements, or other function calls). Surviving FCD mutants help to validate the contract state after a function call, while offering valuable insights into its practical impact and necessity.

M. Function Mutability

Mutations involving modifier deletions (MOD) disclosed many instances where the unhappy path of a function modifier was not adequately tested. Addressing these mutants could also improve the detection of breaking changes, such as the removal of a modifier during code refactoring. Modifier insertions (MOI) can generate benign mutants more frequently, necessitating careful analysis. Nonetheless, similar to the RSI operator, they can help to reason about the contract requirements and identify missing validations. Indeed, one MOI mutant marked as IP patched a vulnerable contract function that was lacking an `onlyOwner` modifier. Yet, access control issues generally require more elaborated mutations to be disclosed, as shown by the limited generation of productive mutants in Table III. One common problem reported by auditors emerges with the usage of OpenZeppelin's Access Control libraries[4]. Due to an implementation oversight, a privileged role can be left vacant after revocation, potentially locking fundamental contract functionalities. The operator set was therefore enhanced as follows:

MOI (Modifier Insertion): The MOI operator was refined to limit the generation of benign mutants. For instance, MOI no longer targets pure or view functions in the contract.

MOC (Modifier Order Change) and MOR (Modifier Replacement): Most mutants generated by MOC during the experiments were equivalent, as they changed the orders of access control modifiers applied to the same function. Since in most cases such mutation does not affect the contract behavior, the operator was discarded. The MOR operator, which substitutes one modifier for another, was excluded as well. Indeed, the effects of its mutations can be replicated through a combination of insertions (MOI) and deletions (MOD), without introducing the risk of masking effects.

RRI (Revoke Role Insertion): The RRI operator targets smart contract that use OpenZeppelin's access control libraries. First, it identifies functions that can only be invoked from a privileged account, based on the specific access control library imported by the contract. For example, the `burn` function in Listing 7 uses the `hasRole` method provided by `AccessControlUpgradeable` to check if the transaction sender has been granted the `ADMIN_ROLE` (line 3). When a viable function is found, RRI inserts two statements to forcefully revoke the transaction sender's privileged role (line 5) and immediately grant it back (line 6). In this way, the mutant can only be killed when the `revokeRole` operation on line 5 fails, and not because its effect are detected by unrelated tests. Mutant survival may indicate that the contract

[3]Example of unspecified evaluation order exploit: https://github.com/ethereum/solidity-underhanded-contest/blob/master/2022/submissions_2022/submission9_TynanRichards/SPOILERS.md

[4]OpenZeppelin Access Control: https://docs.openzeppelin.com/contracts/2.x/access-control

allows the only privileged user to renounce its role. It might also imply that that multiple privileged accounts are set-up before testing, and the single-admin scenario is ignored.

Listing 7: RRI mutation example

```
contract Project7 is AccessControlUpgradeable {
 function burn(address from, uint256 amount)
    ↪ public {
    require(hasRole(ADMIN_ROLE,msg.sender));
    _burn(from, amount);
+   revokeRole(ADMIN_ROLE,msg.sender);
+   _grantRole(ADMIN_ROLE,msg.sender);
 }
}
```

N. Return Statements

By removing the return statement, the RSD operator helped to identify scenarios where the return value of a function was never tested. This allows auditors to assess the necessity of the return value and consider potential code refactoring for improved clarity. However, it is crucial to use RSD in conjunction with other operators, as certain functions affect the contract state without returning any value.

RSD (Return Statement Deletion): Deleting a statement that returns the outcome of a state-changing function is a sub-optimal mutation. This mutant could be killed by unrelated tests that verify the effects of the function call on the contract state, even if the return statement itself is never checked. To avoid this masking effect, RSD now removes the `return` while retaining the original function call (see Listing 8).

Listing 8: Example of improved RSD mutation

```
function withdraw(...) public virtual override
   ↪ returns (uint256) {
 ...
- return super.withdraw(assets, receiver, owner);
+ super.withdraw(assets, receiver, owner);
}
```

O. Ether and Token Transfers

A common issue in audited projects involves unchecked return values from Ether and token transfers. The ETR (Ether Transfer Replacement) operator implemented by SuMo helped to disclose one such case (see Table IV) by swapping `transfer()` and `send()`, but the other injected replacements did not provide useful feedback. Most importantly, its mutation rules did not differentiate between transfers of Ether and tokens, generating uncompilable mutants in the latter case.

TRC (Transfer Return Check): The TRC operator overcomes the limitations of ETR by injecting different mutations based on the type and context of the function call:

a) Reverting Function Calls: It replaces `transfer` with `send` to return false on failure, highlighting missing test coverage for reverting transfers.

b) Standalone Non-Reverting Function Calls: It encloses with a `require` statement those standalone Ether or token transfers that return false on failure. The survival of this mutant can reveal a patch opportunity and encourage the derivation of tests for the failing transfer scenario.

c) Nested Non-Reverting Function Calls: It mutates any nested, non-reverting function call differently depending on its context (see Listing 9). If the success of the transfer is a prerequisite for the execution of some logic, the surrounding statement (e.g., a `require`) is removed. Mutant survival implies that the tests did not adequately cover the failing transfer scenario (e.g., due to insufficient funds). Instead, removing the whole require statement prevents the transfer operation from being executed. In this case, a live mutant reveals that the transfer success and the relative state changes, such as account balances consistency, were not verified. If a variable or a return statement are used to propagate the return value of a transfer, TRC unconditionally returns true, with and without executing the transfer operation.

Listing 9: Example of TRC mutants for nested transfer calls

```
// Require statement
-    require(super.transfer(to,amount));
+    super.transfer(to,amount);
// Return statement
-    return super.transfer(to,amount);
+    super.transfer(to,amount); return true;
// Variable declaration
-    bool success = super.transfer(to,amount);
+    bool success = true;
```

P. Variable Units

The variable unit replacement (VUR) operator could have helped to disclose a gas usage issue, but its rules can be inefficient. Mutations involving Ether units can be overly drastic (e.g., converting `ether` to `wei`), causing compilation errors. Moreover, swapping each time unit with all other possible units generates redundancy.

VUR (Variable Unit Replacement): VUR now replaces each variable unit with its neighbor (e.g., hours with minutes), reducing the likelihood of triggering an out-of-gas exception.

Q. Visibility

Mutations to function and variable visibility keywords (FVR and VVR) were not useful for test improvement and issue disclosure. Meanwhile, FVR ranked as the second highest operator in terms of generated mutants. However, since specific replacements can highlight gas optimization opportunities, FVR and VVR were relocated to a dedicated cluster that auditors can independently enable to focus on code optimizations.

VI. Answering RQ2

In Table V we show the experimental results achieved by the selected projects using the novel mutation strategy. Although the revision of the operators introduced new mutation rules, the optimizations led to a reduction in the total number of generated mutants by 30%, resulting in a time savings of over 114 hours. Meanwhile, the ratio of valid mutants generated by SuMo increased from $84,8\%$ to $92,5\%$, showcasing a more efficient generation process. The enhanced set generated productive mutants for 67% of the issues, a substantial improvement over the original set's 47%. Of these issues, 33% are related to Test-Productive mutants, while

TABLE V: Results for the enhanced SuMo operators

ID	$\#G$	$\#V$	MS	Total MTRI	MTRI with NPM	MTRI with TPM	MTRI with IPM	MTRI with IPM &TPM
1	2578	2341	54,2	6	3	3	0	0
2	425	410	66,4	12	6	5	0	1
3	2411	2218	61,3	9	6	2	1	0
4	1428	1336	46,9	22	5	10	4	3
5	1918	1779	61,1	8	2	2	3	1
6	730	635	77,3	6	2	2	2	0
7	125	125	71,2	3	1	0	1	1
8	3491	3275	48,9	13	1	2	3	7
Tot.	13108	12121	-	79	26	26	14	13

34,2% are related to Inspection-Productive mutants. Notably, the number of issues related to IPM more than doubled after the enhancement of the operators. Table VI shows how the new mutation strategy increased the generation of productive mutants for specific issue types. While the original SuMo operators rarely helped to disclose *unchecked transfers*, with the introduction of the TRC operator all of them could be identified through mutant inspection. A similar scenario can be observed for the *access control* category. The RRI operator played a crucial role in the disclosure of revocable ownership and roles, while the enhanced AVR and NVR operators generated useful mutants for project-specific issues (e.g., see Listing 3). For the *precision loss* category, in the previous experiment two issues were linked to NPM and TPM mutants, respectively. Now, both can be immediately detected through code inspection. Instead, no productive mutants were associated to the *reentrancy* category. This was expected, as even the most elaborate mutations may not effectively target the specific conditions and patterns that underlie such vulnerabilities. Mutation testing also proved generally unsuitable for detecting *missing initialization* and *validation checks*. These issues are characterized by the omission of code, rather than a specific code segment that can be modified through mutation. While certain insertion operators, such as MOI and the novel RSI, did help uncover validation issues, their effectiveness remains context-dependent. Overall, while it remains crucial to complement SuMo with static analyzers and other specialized tools, the value of investing in mutation analysis should not be underestimated. The process of test improvement inherently involves examining mutants, which can also help to reveal flaws in the code under test and support contract refactoring.

VII. Related Work

The usage of mutation testing for improving smart contract test suites has been thoroughly investigated [9], [10], [17]–[20]. However, the relationship between mutants and code inspection activities remains largely unexplored in the blockchain domain. Petrovic et al. [13] conducted a large-scale study on the industrial application of mutation testing involving 1.9 million change sets across four different programming languages. The authors pointed out how a mutant can be considered productive if it is *equivalent* but its analysis advances knowledge and code quality. Petrovic et al. [21] later analyzed over 1500 mutant-related merge requests, shedding light on their value in code inspection and review. While test improvement emerged as the most common approach for resolving mutants, they also observed instances of code refactorings and in-depth discussions about the codebase.

TABLE VI: Productive mutants for each Category of Issue generated by the enhanced SuMo operators

Category of Issue	Total MTRI	MTRI with NPM	MTRI with TPM	MTRI with IPM	MTRI with TPM &IPM
Access Control	7	1	0	4	2
Aliasing	1	0	0	1	0
Block Variable Dependence	1	0	0	1	0
Checks-Effects-Interactions Violation	2	0	2	0	0
Gas usage	4	0	3	0	1
Integer Over/Underflow	1	0	1	0	0
Initialization	3	2	1	0	0
Incorrect Event Log Usage	1	0	0	1	0
Miscalculation	11	2	7	0	2
Logical Oversight	4	1	1	0	2
Other	3	2	1	0	0
Precision Loss	4	0	0	2	2
Reentrancy	3	3	0	0	0
Timing	1	0	1	0	0
Unchecked Transfer	5	0	0	5	0
Validation	28	15	9	0	4
Tot.	79	26	26	14	13

VIII. Threats to Validity

The main threat to the validity of our study concerns the size of the experiment, which is constrained by the computational and manual costs associated with mutation testing. Additionally, the accurate identification of productive mutants requires a meticulous manual analysis of each project and its associated issues. To mitigate this threat we selected projects from various categories, including NFT marketplaces, Governance, DeFi, and others. These were chosen to encompass a range of features and characteristics, with the intention of creating a more diverse and representative subject set.

IX. Conclusions and Future Work

This study assessed the practical usefulness of mutation testing for smart contract auditing. To evaluate the productivity of Solidity mutants, we analyzed a set of projects submitted to an auditing company. Specifically, we investigated whether live mutants could have helped in disclosing issues described in the final security report. Our findings underscore the value of mutation analysis for the code review process. Live mutants can not only enhance test quality but also provide insights into the contract's behavior, potentially revealing issues and prompting code refactoring. Still, the sheer volume of mutants can make it challenging to effectively locate the productive ones and present them in a practical manner to auditors. Future work should introduce a mechanism to omit less interesting mutants [13] and prioritize useful results [22]. In this direction, additional analyses on the severity of issues linked to productive mutants could provide useful prioritization guidelines and lower the cognitive overhead placed on auditors.

Acknowledgements

This research has been supported by [hidden for review].

References

[1] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 615–627.

[2] M. Barboni, A. Morichetta, and A. Polini, "Smart contract testing: challenges and opportunities," in *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2022, pp. 21–24.

[3] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[4] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.

[5] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 435–445.

[6] A. J. Ooutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," 1996. [Online]. Available: https://api.semanticscholar.org/CorpusID:9492620

[7] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[8] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[9] M. Barboni, A. Morichetta, and A. Polini, "Sumo: A mutation testing approach and tool for the ethereum blockchain," *Journal of Systems and Software*, vol. 193, p. 111445, 2022.

[10] J. J. Honig, M. H. Everts, and M. Huisman, "Practical mutation testing for smart contracts," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Cham: Springer International Publishing, 2019, pp. 289–303.

[11] M. Barboni, A. Morichetta, and A. Polini, "Sumo: A mutation testing strategy for solidity smart contracts," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2021, pp. 50–59.

[12] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Does mutation testing improve testing practices?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 910–921.

[13] G. Petrović and M. Ivanković, "State of mutation testing at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, pp. 163–171.

[14] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, "Analysing and comparing the effectiveness of mutation testing tools: A manual study," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 147–156.

[15] S. Phipathananunth, "Using mutations to analyze formal specifications," in *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2022, pp. 81–83.

[16] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just, "An industrial application of mutation testing: Lessons, challenges, and research directions," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 47–53.

[17] E. Andesta, F. Faghih, and M. Fooladgar, "Testing smart contracts gets smarter," in *2020 10th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 2020, pp. 405–412.

[18] P. Chapman, D. Xu, L. Deng, and Y. Xiong, "Deviant: A mutation testing tool for solidity smart contracts," in *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2019, pp. 319–324.

[19] P. H. Hartel and R. Schumi, "Mutation testing of smart contracts at scale," in *International Conference on Tests and Proofs*, ser. LNCS, vol. 12165. Springer, 2020, pp. 23–42.

[20] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen, "Musc: A tool for mutation testing of ethereum smart contract," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1198–1201.

[21] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Please fix this mutant: How do developers resolve mutants surfaced during code review?" in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2023, pp. 150–161.

[22] S. J. Kaufman, R. Featherman, J. Alvin, B. Kurtz, P. Ammann, and R. Just, "Prioritizing mutants to guide mutation testing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1743–1754.