Using Mutation Testing To Improve and Minimize Test Suites for Smart Contracts

Enzo Nicourt
Runtime Verification
France

Benjamin Kushigian*
University of Washington
USA

Chandrakana Nandi
Certora Inc.
USA

Yliès Falcone
Runtime Verification
France

***Abstract*—This paper presents a successful industrial case study on the application of mutation testing to evaluate and improve test suites for smart contracts. ERCx is a comprehensive, hand-written test suite and framework for smart contract testing, created by Runtime Verification. Despite its thoroughness, hand-written tests can miss edge cases. To address this, we employed mutation testing, which introduces small, syntactic changes, known as mutants, to the program. Mutants that go undetected by the test suite highlight its potential weaknesses, and by presenting them as testing goals, mutation testing helps developers iteratively improve their test suites. In this study, we used mutation testing to expand the ERCx test suite with five new test cases, including one potential vulnerability identified as critical by the ERCx developers. We also developed a test redundancy metric by analyzing pairwise correlation of test data on mutants; we used this redundancy metric to minimize the test suite by removing redundant tests. Finally, we ran both the full and minimized test suites on 106 real-world, faulty ERC-20 contracts to compare the suites' effectiveness and efficiency. Our findings reveal that although the minimized test suite has systematically lower running times compared to the full suite, it still detected faults in 105 of the 106 real-world tokens, retaining nearly all of the full suite's fault-detection capability.**

***Index Terms*—Mutation Testing, Smart Contracts, Test minimization**

I. Introduction

In this paper we present an industrial case study on the use of mutation testing to improve and minimize a comprehensive, hand-written test suite for smart contracts.

Smart contracts are programs that carry out financial transactions and run on blockchains like Ethereum [1], Solana [2], and Hyperledger [3]. These programs are written in a variety of languages like Solidity [4], Vyper [5], Rust, etc. Vulnerabilities in smart contracts have been exploited to cause serious financial losses—examples of such exploits are, unfortunately, far too common [6]. The immutability of the blockchain exacerbates the problem—once an error-prone version of a contract is deployed on the blockchain, it cannot be modified. Testing, fuzzing, auditing, and verifying smart contracts are therefore well-studied topics [7]–[18].

Transactions on blockchains are often conducted by exchanging cryptocurrencies, or *tokens*. While Ether is the official token of the Ethereum blockchain, Ethereum allows users to create and exchange their own tokens. According to blockchain explorers [19], there are currently hundreds of thousands of tokens on the Ethereum blockchain with 1,000 tokens created daily and a total market cap in the billions of dollars.

To standardize the process of developing new tokens and facilitate interoperability, the Ethereum Foundation has developed APIs which tokens must implement, along with a set of standards those implementations should comply with. The most common standard is called ERC-20 [20]–[22].

However, standards like ERC-20 are merely *suggestions*, and nothing prevents developers from writing custom API implementations that do not conform to the standards (e.g., for improved performance). This non-conformity often results in faulty token implementations, and these faults can cause serious vulnerabilities.

Over the years, many security vulnerabilities in token contracts have been exploited, (e.g., the infamous DAO attack [23], [24]), and have led to hundreds of millions of dollars in losses.

Rigorous testing prior to smart contract deployment is a simple, effective way to help prevent vulnerabilities. Engineers and researchers at Runtime Verification [13] developed the ERCx test suite for checking the compliance of token implementations with their respective standards along with other correctness properties [1].

Developing a comprehensive test suite is challenging—engineers can only write tests to catch faults they anticipate. But what about tests to prevent faults the developer did *not* anticipate? Mutation testing [25], [26] is an approach for finding these missing tests—it systematically introduces program faults, called *mutants*, that a test suite should be able to detect. Undetected mutants witness potential shortcomings in a test suite (e.g., an unanticipated fault) and can guide a developer to improve the suite by writing tests to detect these faults.

This paper leverages mutation testing to improve the ERC-20 test suite in ERCx. The work was done jointly by teams at Runtime Verification [13], Certora [14], and University of Washington. Despite human effort invested in ensuring the robustness of the test suite, mutants generated by the Gambit tool [2] from Certora Inc., [14] resulted in undetected mutants that ERCx could not detect, exposing weaknesses in the test

* Majority of the work done during an internship at Certora Inc.

[1] https://runtimeverification.com/blog/testing-erc-20-tokens-part-1

[2] https://github.com/Certora/gambit

```
interface IERC20 {
   function totalSupply() external view returns (uint256);
   function balanceOf(address account) external view returns (uint256);
   function transfer(address to, uint256 amount) external returns (bool);
   function allowance(address owner, address spender) external view returns (uint256);
   function approve(address spender, uint256 amount) external returns (bool);
   function transferFrom(address from, address to, uint256 amount) external returns (bool);
}
```

Fig. 1: Key functions in the ERC-20 interface in the Solidity programming language. `address` is a type in Solidity that represents the address of a contract when it is deployed on the blockchain. A `view` function can only read the state of the contract and return a value.

suite. We used these undetected mutants as test goals to strengthen the test suite by adding new tests.

We also developed a test redundancy metric based on tests' ability to detect each mutant, and used this metric to *minimize* the test suite. We compared the full and minimized test suites' effectiveness and efficiency for discovering real-world faults by running both suites on 106 faulty real-world contracts. While the minimized test suite was more efficient than the full test suite, having lower running time for nearly all contracts, it still detected faults in 105 of 106 contracts, showing that it is nearly as effective at real-world fault detection as the full test suite. The modified ERCx test suite is publicly available for the Solidity developer community [3] . Our methodology for improving test suites can be extended to support other smart contract standards like ERC-4626 and ERC-1155.

In summary, the contributions of this paper include:

- Using mutation testing to find five missing tests in a robust, industry scale smart contract testing framework, ERCx.
- Leveraging a redundancy metric based on tests' ability to detect mutants to minimize test suites.
- An evaluation on 106 real-world, faulty ERC-20 tokens showing that the minimized test suite is more efficient than the full test suite while retaining nearly all of the full suite's efficacy.

The rest of the paper is organized as follows: Section II offers background on ERC-20 and mutation testing, Section III presents the ERCx test suite, Section IV describes how we used mutation testing to add new tests to the ERCx test suite, Section V discusses our results and the minimization algorithm, Section VI presents key lessons learned, limitations and future work, Section VII discussed relevant related work, and Section VIII concludes.

II. Background

This section offers background on the standard ERC-20 interface and mutation testing. The work in this paper targets the Solidity programming language [4] which is one of the most commonly used languages for developing smart contracts [4] for the Ethereum blockchain [1].

[3]https://ercx.runtimeverification.com

[4]https://chain.link/education-hub/smart-contract-programming-languages

A. ERC-20

ERC-20 is a standard [20] for enabling interoperability across different tokens. As part of the standard, the ERC-20 proposes an API that all tokens must implement to be considered an ERC-20 token. Figure 2 shows the ERC-20 interface in Solidity. The API consists of six core functions: The `totalSupply` function returns the total number of tokens in circulation. The `balanceOf` function allows users to check the balance of a specific account. The `transfer` function enables the transfer of tokens from one account to another. The `transferFrom` function allows a third party to transfer tokens on behalf of an account that has approved them to do so. The `approve` function allows an account to approve another account to spend a certain number of tokens. The `allowance` function allows an approved account to check the amount of tokens they can spend on behalf of another account. These are all the *external* functions of the contract, but they often invoke other functions that are *internal*, which means that they can only be called by other functions from within the contract itself. A view function (`allowance` and `balanceOf`) indicates that it can only read the storage of the contract but not write to it. `address` is a Solidity datatype that represents a 20 byte Ethereum address. More details about the qualifiers and types can be found in the Solidity documentation [4].

B. Mutation Analysis and Testing

Mutation analysis [28], [29] is an automated technique that uses systematically generated program faults called *mutants* to evaluate a test suite's efficacy. The faults are typically small, syntactic changes (e.g., changing operators, changing literal values, altering branch conditions). The test suite is run on the generated mutants, each acting as a proxy for a real-world bug. We say that a mutant is *detected* by a test suite if it fails some test that the original program passed. Stronger test suites should, on average, detect more bugs than weaker test suites and mutation analysis captures this with the *mutation score* metric, which is defined to be #(detected mutants) / #(generated mutants). Stronger test suites correspond to higher mutation scores.

Mutation testing [25], [26] is a technique where a developer is presented with undetected mutants as test goals; the developer creates new tests to detect these mutants. The goal of mutation testing is not necessarily to get a perfect

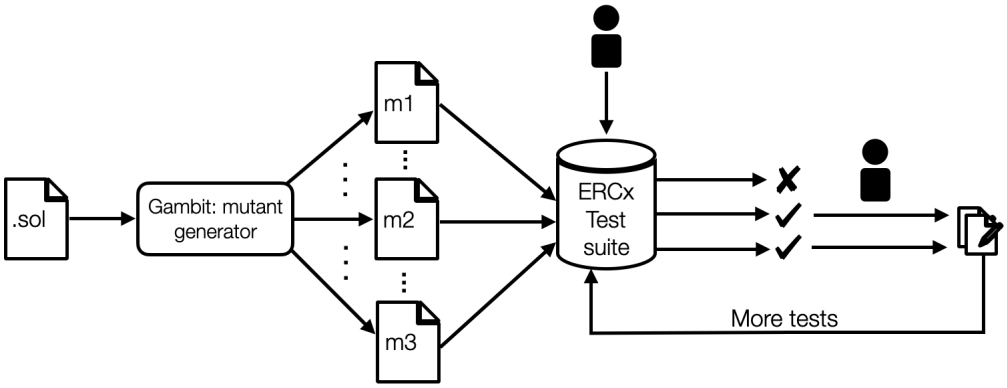

Fig. 2: Workflow for using mutants to develop new tests—we started with ERCx's manually written test suite. We used mutation testing to synthesize faulty versions of a canonical implementation of ERC-20 [27]. We then ran the test suite on all the mutants; we added new tests to the test suite based on any undetected mutants that passed all the original, hand-written tests.

mutation score, which is often infeasible. Rather, mutation testing iteratively strengthens a test suite over time.

III. ERCx Design Principles

ERCx is an extensible testing framework created by Runtime Verification [13] and designed for Ethereum-based tokens. Currently, it supports testing of ERC-20 tokens, one of the most common tokens in the Ethereum blockchain [19]. Developers are actively extending ERCx to support additional standards like ERC-4626, ERC-721, etc. ERCx executes *property tests* [30], that is, tests that directly evaluate properties (e.g., invariants) or contract features. ERCx categorizes tests into five main categories: *critical*, *recommended*, *desirable*, *fingerprint*, and *ABI*.

The critical category contains tests that cover the basic requirements of the ERC-20 standard and *must* be satisfied. Tokens that pass the *critical* tests comply with the ERC-20 standard. The *recommended* category includes tests that cover more advanced features of the ERC-20 standard, such as the allowance mechanism, events, and token-burning functionality. These tests are recommended for tokens that wish to go beyond the basic requirements of the ERC-20 standard and offer additional functionality to their users. The *desirable* category includes tests that cover features beyond compliance and are desirable for tokens seeking to provide a more robust user experience and higher security. These tests cover features such as gas optimization, reentrancy protection, and token recovery functionality. The *fingerprint* category includes tests that are specific to certain implementation choices. The *ABI* category includes tests that check whether a token conforms to the ERC-20 specification regarding the Application Binary Interface (ABI). By categorizing tests in this way, ERCx provides a set of tests that covers a wide range of token functionality, from the basic requirements of the ERC-20 standard to more advanced features and desirable functionality. This test suite was designed to adequately represent the ERC-20 specification and indicates a best-effort attempt to capture the constraints of ERC-20.

ERCx Execution Model. To run its tests on a deployed smart contract, the ERCx framework creates a fork of the Ethereum blockchain environment using the Forge tool [31] by establishing a remote procedure call (RPC) endpoint. ERCx interacts with the latest state of the blockchain through the RPC to query and alter the state of the contract during testing. For example, it may query for a balance or initiate a token transfer. Connecting to the blockchain through an RPC can introduce variability in the running times of ERCx.

IV. Methodology

A. Test Selection

We filtered the ERCx test suite to only retain those from the *critical*, *recommended*, and *desirable* categories, since these are the categories that are most important for functional correctness and security of the contracts. This led to 57 tests: 18 *critical*, 2 *recommended*, and 37 *desirable*.

B. Selecting a Canonical ERC-20 Reference Implementation

Mutation analysis requires that all tests in a test suite pass when run on the original program: if there are faults in the original program then a test failing on a mutant may be due to a fault in the original program rather than the test's ability to detect the fault introduced by the mutant. Thus it is critical to perform mutation analysis on a high-quality canonical reference implementation of the ERC-20 contract.

We selected OpenZeppelin's publicly available implementation of ERC-20 as our canonical reference implementation [27]. The OpenZeppelin implementation, which is comprised of 316 LoC of Solidity, has been heavily audited by security experts, and it has had many important correctness properties formally verified [14], [32].

$\langle gambit\_conf \rangle$ ::= $mutation\_task^+$

$\langle mutation\_task \rangle$ ::= *filename solc? contract? function* mutation_op* solc_arg**

$\langle filename \rangle$ ::= string $\langle contract \rangle$ ::= string

$\langle function \rangle$ ::= string $\langle solc \rangle$ ::= string

$\langle mutation\_op \rangle$ ::= BOM | UOM | RSM | ASM | DEM | ISM | SOM | SLM | EDM

$\langle solc\_arg \rangle$ ::= optimize | remappings | ...

Fig. 3: Syntax of the core subset of Gambit's configuration language. The mutation operators are defined in Section IV-C. *sol_arg* supports a variety of arguments supported by various Solidity versions.

C. Mutant Generation

Gambit[5] is an open-source, state-of-the-art mutant generator for Solidity smart contracts, developed by Certora [14]. Gambit mutates Solidity source code by traversing the program's Abstract Syntax Tree (AST) and applying predefined rules, called *mutation operators*, to alter the program's syntax. Mutants generated by Gambit are *first order* [33], i.e., each mutant has a single change. Gambit uses a standard set of mutation operators that are inspired by prior work [34]–[36].

Gambit only *generates* mutants; Gambit's client decides how the mutants will be used. For example, the mutants can be used to evaluate test suites or formal specifications. In this paper we used Gambit to evaluate ERCx's test suite for ERC-20 tokens. As another example, at Certora, Gambit is integrated with the formal verification tool, CVT [37]. Gambit is actively being developed—in this paper we used commit 88e145b of the tool. This version of Gambit supports the following mutation operators:

- *Binary Operator Mutation (BOM)*: Replace binary operators, $+, -, \times, /, **, \%$ with a different operator.
- *Unary Operator Mutation (UOM)*: Replace unary operators, $++, --, \sim$ with a different operator.
- *Require Statement Mutation (RSM)*: Replace the condition of a `require` statement with `true` and `false`.
- *Assignment Statement Mutation (ASM)*: Change the right-hand side of a Boolean assignment statement with `true` and `false`, and the right-hand side of an integer assignment statement with $-1, 0, 1$.
- *Delete Expression Mutation (DEM)*: Comment out an expression.
- *If Statement Mutation (ISM)*: Replace the condition of an `if` statement or change it to `true` and `false`.
- *Swap Operator Mutation (SOM)*: Swap the operands of a non-commutative binary operator.
- *Swap Lines Mutation (SLM)*: Swap two consecutive lines of code.
- *Eliminate Delegate Call Mutation (EDM)*: Replace a `delegatecall` by a `call` method in Solidity. The main difference between the two is whether the function is called in the context of the caller or the callee [6].

Gambit offers a configuration language that allows users to customize the kinds of mutations and localize them to specific functions or contracts in the code. The grammar for the language is shown in Figure 3.

A Gambit configuration program is a list of configurations, each of which defines a mutation task. A mutation task must include a *filename* and optionally other arguments. Gambit ensures that it produces valid mutants by running the Solidity compiler on each mutant and filtering out invalid ones. Gambit allows the user to configure the Solidity compiler by passing various flags supported by Solidity compilers like optimization settings, import paths, and remappings. Gambit also allows users to *localize* mutants to specific contracts and functions, and lets the user control which mutation operators they want to enable. These features help Gambit generate fewer unhelpful mutants. Using these filters, we avoid mutating internal functions [7] that are never invoked by any public function. There were two such functions, `_burn` and `_mint`. In total, we obtained 64 mutants in 6 seconds using the following configuration:

```
{
    "filename": "ERC20.sol",
    "solc": "solc8.20",
    "contract": "ERC20",
    "functions": [
        "name",
        "symbol",
        "decimals",
        "totalSupply",
        "balanceOf",
        "transfer",
        "_transfer",
        "_spendAllowance",
        "_update",
        "allowance",
        "approve",
        "_approve",
        "transferFrom"
    ]

}
```

Upon manual inspection, we further found that 16 of these mutants had changes in internal functions (like `_update`) that are only be observable from other internal functions and therefore cannot be detected by ERCx. We removed these additional tests and performed all our experiments with 48 mutants.

V. Evaluation and Analysis

To evaluate the benefits of mutation-guided testing of smart contracts, this section answers three research questions.

RQ1 Can undetected mutants be used to increase test adequacy [38] of comprehensive hand-crafted test suites (Section V-A)?

[5]https://github.com/Certora/gambit

[6]https://solidity-by-example.org

[7]https://docs.soliditylang.org/en/latest/contracts.html#visibility-and-getters

```
function _spendAllowance(address o, address s, uint256 a) internal virtual {
    uint256 curr = allowance(o, s);
    if (curr != type(uint256).max) {
        // ...
        // ORIGINAL: _approve(o, s, curr - a, false);
        _approve(o, s, curr + a, false); // MUTANT
    }
}
```

(a) M1 increases, instead of decreases, an address's allowance

```
function _transfer(address f, address t, uint256 value) internal {
    // ORIGINAL: if (f == address(0)) {
    if (false) { // MUTANT
        revert ERC20InvalidSender(address(0));
    }
    // ...
}
```

(b) M2 skips a condition check for the zero address

```
function _approve(address o, address s, uint256 a, bool e) internal virtual {
    // ORIGINAL if (o == address(0)) {
    if (false) { // MUTANT
        revert ERC20InvalidApprover(address(0));
    }
    // ...
}
```

(c) M3 skips a condition check for the zero address

```
function _approve(address o, address s, uint256 a, bool e) internal virtual {
    // ...
    // ORIGINAL if (s == address(0)) {
    if (false) { // MUTANT
        revert ERC20InvalidSpender(address(0));
    }
    // ...
}
```

(d) M4 skips a condition check for the zero address

```
function _spendAllowance(address o, address s, uint256 a) internal virtual {
    uint256 curr = allowance(o, s);
    // ORIGINAL if (curr != type(uint256).max) {
    if (true) { // MUTANT
        if (curr < a) {
            revert ERC20InsufficientAllowance(s, curr, a);
        }
        unchecked {
            _approve(o, s, curr - a, false);
        }
    }
}
```

(e) M5 skips a condition check to ensure an allowance is not `type(uint256).max`

Fig. 4: Five mutants from Gambit that passed all the hand-written tests in ERCx, exposing weaknesses in the test suite. Each row shows the original function body (pink highlight shows original) and the mutated function body (green highlight shows mutant) generated by Gambit. The full code of the original contract can be obtained from Openzeppelin [27] as mentioned in Section IV-B. Functions starting with _ (like `_approve`, `_update`) are internal to the contract (Section II-A)

.

RQ2 Does a mutant-based correlation between tests correspond to a redundancy between tests, and can this redundancy be used to minimize a test suite without affecting test suite adequacy (Section V-B)?

RQ3 Is a minimized test suite as effective at real-world fault detection as the original test suite (Section V-C)?

We conducted all the experiments on a machine with 32GB of RAM, 10 cores and running macOS Ventura 13.3.1. The scripts for reproducing the graphs, data, and plots can be found here: https://github.com/Certora/gambit/tree/icst2024/icst2024.

A. RQ1: Increasing Test Adequacy from Undetected Mutants

We ran 57 original, hand-written tests from ERCx on all 48 mutants. This original test suite detected 43 of the 48 generated mutants (mutation score of 89.6%). The remaining five mutants, shown in Figure 4, were not detected:

- M1 modifies `_spendAllowance()` to increase, instead of decrease, an address's approved allowance when that address spends tokens.
- M2, M3, and M4 replace checks that ensure that the zero address [39] is never participating in a transaction.
- M5 eliminates a condition check in `_spendAllowance()` that ensures that the current allowance is not the maximum value `type(uint256).max`.

Tests from Undetected Mutants. Each of the undetected mutants M1–M5 corresponds to a potential fault in an ERC-20 implementation that would not be detected by the original hand-crafted test suite. We therefore extended the test suite with five new test T1–T5 which detect M1–M5 respectively; we refer to this extended test suite as the *full test suite*, T_{Full}.

We illustrate this process by examining the test T1 that was written to detect M1. M1 mutates `_spendAllowance()` by replacing `_approve`'s argument `currentAllowance - a` with `currentAllowance + a`. These two expressions evaluate to different values iff `a` is non-zero, so any test that detects this mutant must provide a non-zero value for `a`. `_spendAllowance()` is an internal function and a test cannot invoke it directly. Instead, a test must invoke the public API in a way that will cause the fault to be executed. `transferFrom()` is the only external function that calls `_spendAllowance()`:

```
function transferFrom(address f, address t, uint256 v)
    public virtual returns (bool) {
    address s = _msgSender();
    _spendAllowance(f, s, v);
    _transfer(f, t, v);
    return true;
}
```

`transferFrom()` passes its `v` argument as the `a` parameter of `_spendAllowance` (Figure 4), and the mutated expression will differ from the original expression iff `v` is non-zero. A simplified version of test `testTransferDecreasesAllowance` from ERCx is given below:

```
function testTransferDecreasesAllowance(
            uint256 am, uint256 all)
    assume(am > 0);
    assume(am <= balanceOf(alice));
    assume(all >= am);
    assume(all < MAX_UINT256);
    assertSuccess(alice.approve(bob, all));
    assertSuccess(bob.transferFrom(alice, carol, am));
    assertEq(all - am, allowance(alice, bob));
}
```

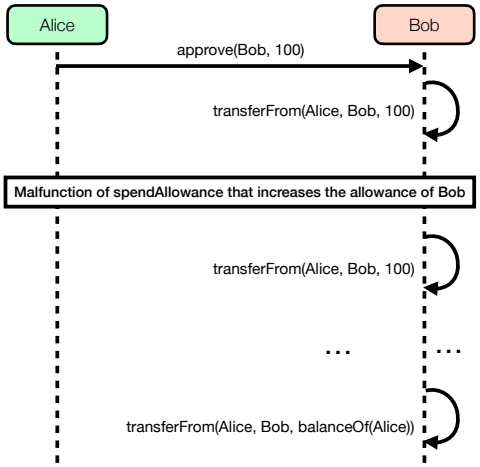

Fig. 5: Sequence diagram showing how a malicious user (Bob) can take advantage of the vulnerability exposed by the first mutant in Figure 4.

TABLE I: Tests added to ERCx based on undetected mutants and the ERCx category it is assigned.

ID	Test Name	ERCx Category
T1	`testTransferDecreasesAllowance`	Critical
T2	`testZeroAddressCannotTransfer`	Recommended
T3	`testZeroAddressCannotApprove`	Recommended
T4	`testCannotApproveZeroAddress`	Recommended
T5	`testInfiniteApprovalConstant`	Fingerprint

Mutants M2–M5 led to tests T2–T5 in a similar fashion. Tests T2–T4 catch instances of the zero address vulnerability [39], and T5 checks that infinite allowance of a user always remains infinite and does not ever change. Note that T5's relevance depends on the implementation of the ERC-20, and failure of this test does not necessarily indicate a bug—it might however point the developer to other potential faults in the code.

Table I summarizes the new test cases. The ERCx developers labeled test T1–T5 according to the schema introduced in Section III. Test T1, which prevents a real vulnerability presented below (Section V-A1, was labelled *critical*, T2–T4 were labeled *recommended*, and T5 was labeled Fingerprint.

For the rest of the evaluation, we discard T5 as it is not in the recommended, critical, or desirable category. In summary, we ended up with 61 relevant tests in ERCx after improving it with mutation testing: 19 critical, 5 recommended, and 37 desirable (did not change).

1) Attacking M1: To understand the importance of T1, consider the attack scenario (Figure 5) that exploits the vulnerability introduced by M1. Alice approves some allowance for Bob (in this case, 100 tokens), and Bob uses `transferFrom()` to transfer this allowance to himself. A correct implementation of the ERC-20 contract would decrement Bob's allowance to 0,

and Bob would not be able to transfer further funds from Alice to himself. Instead, M1's fault *increases* Bob's allowance from 100 tokens to 200 tokens, and Bob can continue to withdraw funds from Alice. *We conclude that uncaught mutants can act as new test goals that can be used to improve existing test suites.*

B. RQ2: Using Test Correlation for Test Suite Minimization

Test suites face a trade-off between effectiveness and resource usage (usually measured in test suite running time). Test suite minimization is as technique to reduce a test suite's resource usage without reducing the suite's effectiveness. To answer RQ2 we ran the full test suite, T_{Full}, from Section V-A on all mutants, and used the results to find redundancies between tests. We then removed redundant tests manually, resulting in a minimized test suite T_{Min}.

A test enforces program properties (e.g., withdrawing tokens should decrease an address's allowance). Once a program property has been tested, additional tests offer diminishing returns: the first test will already be detecting some potential faults, and additional tests are less likely to detect new faults. We say that two tests are *redundant* if they enforce similar program properties.

Mutants correspond to an alteration of some program property or behavior: if multiple mutants alter the same property then they are more likely to be detected by tests enforcing that property. Thus, redundant tests should detect similar sets of mutants and non-redundant tests should detect dissimilar sets of mutants, and we use this insight to develop a proxy measure for test redundancy.

1) Building a Test Correlation Model: Let T be a test suite of program P, and let M be a mutant set for P. T's *kill matrix* [40] K for mutant set M is defined to be the $|T| \times |M|$ matrix, indexed by tests T and mutants M, such that for test $t \in T$ and mutant $m \in M$:

$$K[t, m] = \begin{cases} 1 & \text{if test } t \text{ detected mutant } m \\ 0 & \text{if test } t \text{ did not detect mutant } m \end{cases}$$

K's rows correspond to a single test's performance across all mutants, and we denote the tth row of K by $K[t]$.

Given a kill matrix K for T and M we can compute the correlation between two tests t_i and t_j by computing the correlation of their rows $\text{corr}(K[t_i], K[t_j])$. High correlation between two tests indicate that the tests detected similar sets of mutants, and low correlations indicate that the tests detected different sets of mutants. We define the *correlation matrix* C of T with respect to M to be the matrix that maps pairs of tests to their correlation:

$$C[t_i, t_j] = \text{corr}(K[t_i], K[t_j]).$$

We computed the kill matrix K_{Full} of T_{Full} for the mutant set generated in Section IV-C, and used this to compute the correlation matrix C_{Full} of T_{Full} with respect to that mutant set. C_{Full} is shown in Figure 7 (Left). The X and Y axes represent the 61 tests of T_{Full}. Redder cells indicate higher correlation, bluer cells correspond to lower correlation, and white cells indicate no correlation. The matrix is symmetric, since $C_{\text{Full}}[t_i, t_j] = C_{\text{Full}}[t_j, t_i]$, and it's down diagonal, which plots the correlation of all tests with themselves, is uniformly 1, showing that all tests are perfectly correlated with themselves.

```
def h_minimize(T, corr, h):
    """
    Greedily compute an h-minimization of
    test suite T given test correlations
    corr and threshold h
    """
    i = 0
    while i < len(T):
        j = i + 1
        while j < len(T):
            if corr(T[i], T[j]) >= h:
                T.pop(j)
            else:
                j += 1
```

Fig. 6: Greedy minimization algorithm for eliminating redundant tests.

2) Using the Correlation Model to Minimize T_{Full}: For test suites T, $S \subseteq T$, and threshold $h \in [0, 1]$, we say that S is an h-minimization of T if S is a maximal set with the property that $\text{corr}(t_i, t_j) < h$ for all distinct $t_i, t_j \in T$. Formally,

1) $\forall t_i,\ t_j \in S,\ t_i \neq t_j \implies \text{corr}(t_i, t_j) < h$ and
2) $\forall t_i \in T \setminus S,\ \exists t_j \in S,\ \text{corr}(t_i, t_j) \geq h$.

We construct an h-minimization of T with the greedy algorithm shown in Figure 6.

T_{Full} took 561s to run on over all 48 mutants. To study the effect that minimization has on the tradeoff between suite running time and effectiveness, we used `h_minimize` to compute h-minimizations of T_{Full} for h values from 0.95 to 0.5 in increments of 0.05. This resulted in 10 new test suites: $T_{0.95}$, $T_{0.90}$, ..., $T_{0.55}$, and $T_{0.50}$. We ran each h-minimization on the mutants and recorded each suite's running time and number of undetected mutants (Figure 7 (Right)). $T_{0.90}$ maintained perfect effectivness, detecting all mutants, while also seeing a 50% reduction in the running time from $T_{0.95}$. However, lower h values began to lose effectiveness without any meaningful decrease in running time. Thus we chose $T_{0.90}$ as the minimized test suite, and we refer to it as T_{Min} for the remainder of this paper. T_{Min} contains 34 tests, 27 fewer than the 61 in T_{Full}. Notably, these 27 tests did not include any of the 5 new tests added in Section V-A, suggesting that these new tests enforce properties not enforced by other tests in T_{Full}. *We conclude that the test correlation model can be effectively used to minimize test suites.*

C. RQ3: Running a Minimized Test Suite on Real, Buggy Contracts

In Section V-B we saw that minimized test suite T_{Min} is able to detect all generated mutants. However, minimization is only useful if it does not affect the suite's ability to detect

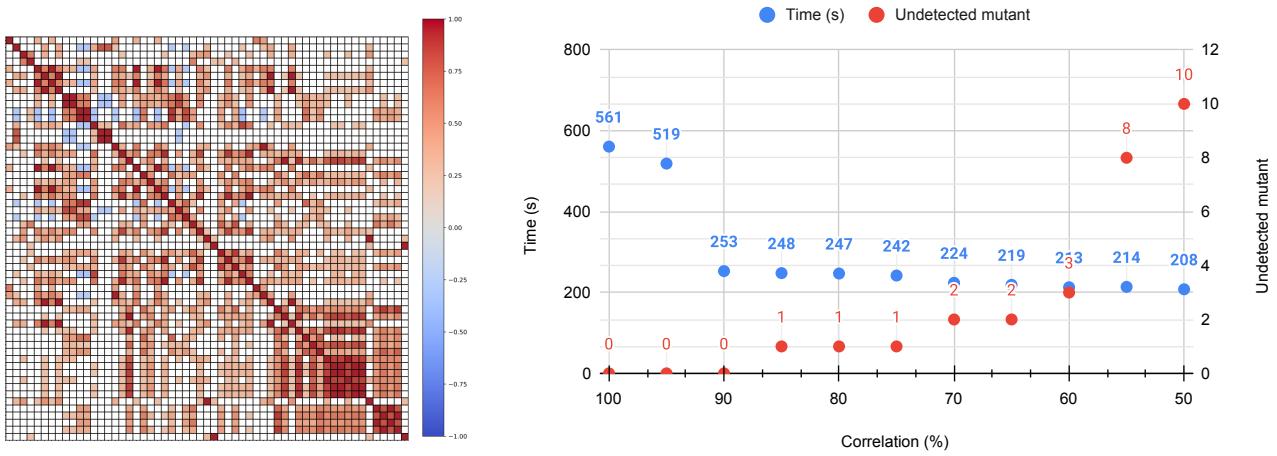

Fig. 7: (Left) Correlation between ERCx tests across 48 mutants from Gambit. Both X and Y axes represent the 61 tests, including the five added based on mutations. Red means higher correlation, blue means lower correlation. (Right) Relationship between total running time of each test suite across all 48 mutants and the number of mutants caught, as correlated tests are gradually removed. We see that when one of two highly correlated tests (above 90%) are removed, the coverage of the test suite in terms of mutants caught remains the same (100%), but the running time of the test suite decreases from 561 seconds to 253 seconds, or 5.3 seconds per mutant.

real-world faults. In this section we investigate the efficacy of T_{Min} for detecting faults in real-world token contracts.

We used the awesome-buggy-erc20 [41] dataset, a collection of 107 real-world, faulty ERC-20 tokens deployed on the Ethereum blockchain. We obtained source code for all 107 contracts from Etherscan [19]. We discarded one contract due to long running time, leaving 106 real-world buggy contracts. We ran both the full test suite T_{Full} of 61 tests and the minimized test suite T_{Min} of 34 tests on all 106 contracts and recorded the number of failed tests from each suite on each contract.

T_{Full} detected a fault in all buggy contracts, and T_{Min} detected a fault in all but one of the buggy contracts. The bubble plot in Figure 8 shows the failure rates of T_{Full} and T_{Min} for each of the real-world contracts. Each bubble is close to the $y = x$ line, indicating that the failure rates for both test suites are approximately the same across all contracts. This implies that the minimization process removes tests in an unbiased fashion with respect to real world faults.

Figure 9 shows the running times of T_{Full} and T_{Min} on each of the 106 contracts. T_{Min} saw consistent speedups for all but two contracts (DGX and LGD). This is due to network noise as described in Section III.

We conclude that T_{Min}, which was minimized using a mutant-based test correlation metric, retained nearly all of T_{Full}'s real-world fault detection capabilities.

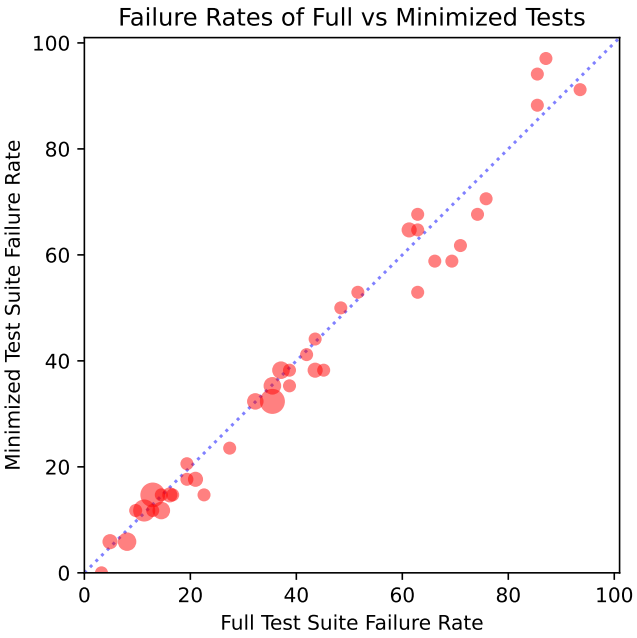

Fig. 8: Each dot represents the results of running full test suite T_{Full} and minimized test suite T_{Min} on one or more real-world faulty ERC-20 contracts. A dot's X position shows T_{Full}'s failure rate on the corresponding contracts; a dot's X position shows T_{Min}'s failure rate on the corresponding contracts. Larger dots correspond to more contracts with the same failure rates for both test suites.

VI. Discussion

This section discusses lessons learned, threats to validity, limitations, and future directions we are interested in pursuing.

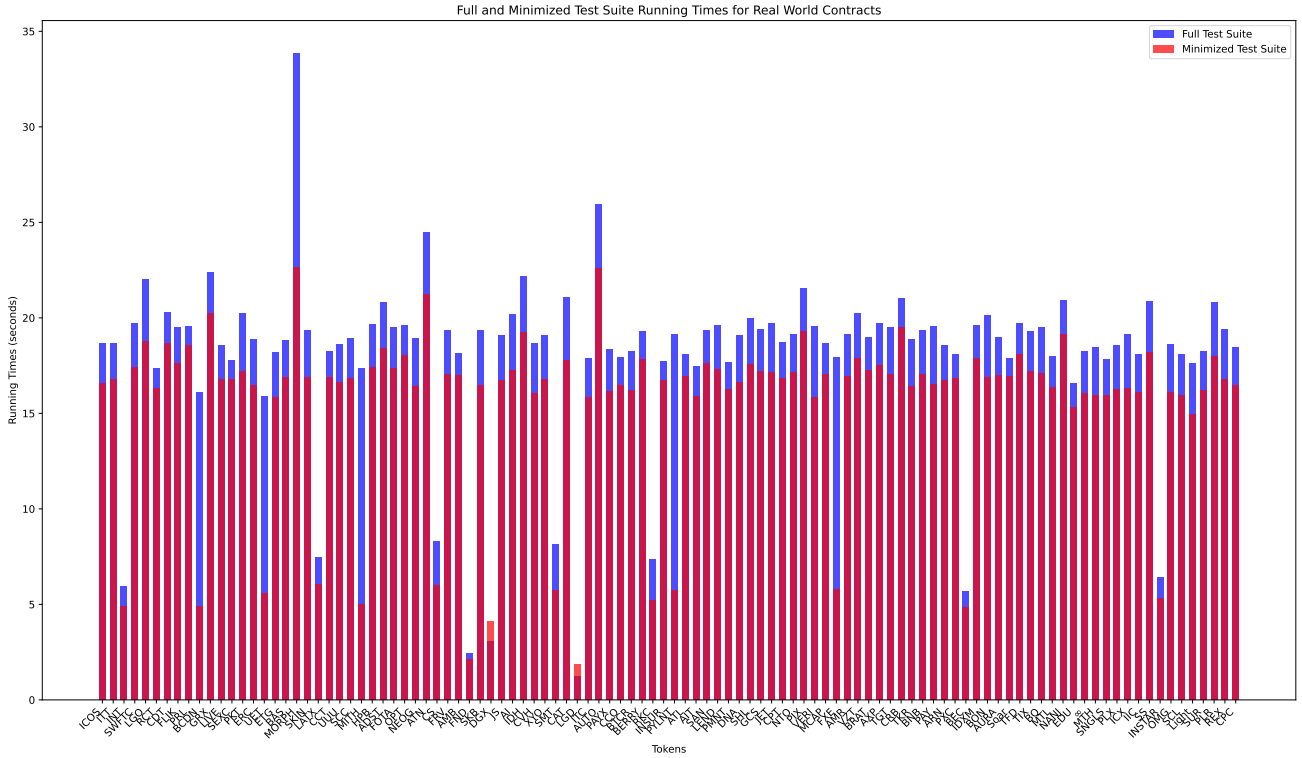

Fig. 9: Comparing the running times of the two test suites T_{Full} and T_{Min} on 106 real-world, buggy contracts. The names on the x-axis represent the symbols of the tokens. Each bar is the average of three runs of the test suite. Due to network noise, T_{Min} ran longer than T_{Full} on two tokens (DGX and LGD).

A. Lessons Learned

This paper has three key takeaways that are applicable to other smart contracts and also generalize to programs beyond smart contracts.

Mutation-testing finds missing tests in robust test suites. Even for thoroughly designed hand-written test suites, mutation-guided test generation is an effective way to find weaknesses. Hand-written tests may fail to detect faults that test writers did not consider, and mutation testing elicits new tests that target the faults a test writer may not have considered.

Mutants can be used for test minimization. A correlation model between tests, computed by running the tests on mutants, can be used to minimize a test suite that retains most of the full test suite's adequacy. We believe that this correlation model is not unique to smart contracts and can be generated for any domain using mutants and can be leveraged to prioritize tests and minimize test suites.

Mutants reflect test behavior on real contracts. A test suite can be minimized by removing tests that are highly correlated with other tests, where correlation is measured by two tests' ability to detect / not detect a mutant. We found that such a minimized test suite is still effective in catching bugs in real-world smart contracts.

B. Threats to Validity

Mutant generation is a simple syntactic fault generation technique and does not consider program context or program semantics. This means that there may be *redundancy* in the mutant set: many mutants might correspond to a single fault [42], [43]. This in turn could possibly affect correlations between tests and alter test minimization. We believe that any such affect will be minor: tests that agree on redundant mutants still agree, and while the exact level of correlation may fluctuate, the amount of fluctuation should be relatively small.

C. Limitations and Future Directions

In this work, we evaluated, improved, and minimized a test suite for ERC-20 token contracts. The canonical implementation we chose is only several hundred lines long, and this made obtaining a perfect mutation score feasible. Smart contracts for other types of tokens are comprised of many more lines of code which will lead to many more mutants—running an entire test suite on mutants for larger programs would consume more resources. Making a comprehensive test suite guided by mutants would therefore be more challenging, though still feasible, for larger programs. However, this is a one-time cost payed by the test developer while writing tests; token designers do not need to pay this cost once the test suite is written.

Additionally, redundant mutant detection can be used to simplify the mutant set: by removing mutants that correspond to the same underlying fault we reduce the number of test suite invocations. Additionally, future work could leverage techniques from mutation testing literature [42], [44] to mitigate this problem.

Nevertheless, the insights we obtained from the correlation model and the technique of the mutation-guided test suite generation are general and we are currently exploring how to apply them for other standards like ERC-721, ERC-4626, ERC-1155, etc. The core ideas of this paper are not specific to smart contracts and are applicable to areas beyond decentralized finance, and prior work has already explored these directions in other contexts (Section VII).

VII. Related Work

Mutation testing and analysis are well-studied topics, in academia [45] and industry [46], [47]. Mutation analysis [28], [29] has been used to study the efficacy of test suites in terms of number of detected mutants. Mutation testing [47] has also been explored for improving test suites by adding tests guided by undetected mutants. Recent surveys [48], [49] provide a thorough discussion of both.

A. Mutation Testing for Smart Contracts

In the context of smart contracts, several mutation analysis tools have been developed [34], [35], and more generic mutation tools like the UniversalMutator [36] have been extended to support smart contract languages like Solidity [4]. These tools present early results of using mutation testing to evaluate the coverage of test suites for a small number of smart contract projects. Vertigo [35] uses mutations to show gaps in the test suites from two smart contracts. To the best of our knowledge, this is the first work that uses mutations to (1) discover new tests in robust, industry-scale test suites, and (2) generate reduced test suites based on a correlation model, in the domain of smart contracts.

B. Correlation Models and Test Suite Minimization

Mutation testing based correlation modeling [50] has been explored in the context of Java programs to understand if mutants are suitable replacements for real faults. Test minimization is a well-studied topic [51], [52]. Prior work has explored the use of greedy algorithms to select test cases that have the most coverage in terms of lines of code. Prior work also used mutants to develop multi-objective-optimization solutions for test prioritization [53]. A recent survey [54] presents several techniques used for test minimization and the use of mutation testing to study the efficacy of the minimized test suites. However, unlike this paper, they only use artificial bugs (mutants) to study efficacy. Chen et al. [38] studied the relationship between test set size, fault detection, and various criteria used to measure the adequacy of a test suite, like coverage, mutation score, etc. The work in this paper is similar to Chen et al.'s idea of reducing test suites based on mutation adequacy, which the authors show to be better than measures like coverage adequacy. We showed that a test suite minimized based on a correlation model guided by mutants can effectively find bugs in real-world programs.

C. Other Methods for Checking Smart Contracts

This paper focuses on testing smart contracts and using mutation testing to improve test suites. There are many other techniques used to check and / or guarantee the correctness of smart contracts. Formal verification [9]–[11], [14], [16], [17] has been used to check that smart contracts satisfy important properties. Many fuzzing tools [15], [55]–[57] both in industrial and research settings have been proposed for testing smart contracts on concrete inputs. Recent surveys [44], [58] cover a wide range of tools and techniques for verifying and testing smart contracts.

VIII. Conclusion

This paper presents a successful industrial case study on the application of mutation testing to evaluate and improve test suites for smart contracts. We found that undetected mutants led to the discovery of five new tests that were missed by experts, including one labeled *critical* by developers. Tests that detected similar mutants indicated test redundancy, and we used this redundancy to minimize the test suite. We showed that this minimized test suite was able to detect faults in 105 out of 106 faulty, real-world smart contracts. Going forward, we are eager to apply this approach to other protocols, other languages for smart contracts, and to other domains beyond smart contracts.

ACKNOWLEDGEMENTS

We thank the teams at Certora Inc. and Runtime Verification for their help with developing Gambit and ERCx. We also thank the anonymous reviewers for their thoughtful comments. We are grateful to René Just and Zachary Tatlock from University of Washington for their feedback on earlier versions of this work.

References

[1] "Welcome to ethereum," 2023, https://ethereum.org.
[2] "Solana. powerful for developers. fast for everyone." 2023, https://solana.com/.
[3] "Hyperledger foundation." 2023, https://www.hyperledger.org/.
[4] "Solidity documentation," 2023, https://docs.soliditylang.org/en/v0.8.20/.
[5] "Vyper documentation," 2023, https://docs.vyperlang.org.
[6] "Rekt leaderboard," 2023, https://rekt.news/leaderboard/.
[7] Y. Nishida, H. Saito, R. Chen, A. Kawata, J. Furuse, K. Suenaga, and A. Igarashi, "Helmholtz: A verifier for tezos smart contracts based on refinement types," 2021. [Online]. Available: https://arxiv.org/abs/2108.12971
[8] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 454–469. [Online]. Available: https://doi.org/10.1145/3385412.3385990
[9] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," *CoRR*, vol. abs/1711.09327, 2017. [Online]. Available: http://arxiv.org/abs/1711.09327

[10] A. Mavridou, A. Laszka, S. Emmanouela, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum," in *Proceedings of the 23nd International Conference on Financial Cryptography and Data Security (FC)*, February 2019.

[11] S. Wesley, M. Christakis, J. A. Navas, R. Trefler, V. Wüstholz, and A. Gurfinkel, "Compositional verification of smart contracts through communication abstraction," in *Static Analysis*, C. Drăgoi, S. Mukherjee, and K. Namjoshi, Eds. Cham: Springer International Publishing, 2021, pp. 429–452.

[12] Certik, "Certik," 2022, https://www.certik.com/.

[13] RV, "Runtime verification," 2022, https://runtimeverification.com/.

[14] Certora, "Certora prover documentation," 2022, https://docs.certora.com/en/latest/.

[15] B. Jiang, Y. Liu, and W. K. Chan, *ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection*. New York, NY, USA: Association for Computing Machinery, 2018, p. 259–269. [Online]. Available: https://doi.org/10.1145/3238147.3238177

[16] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng, "Soltype: Refinement types for arithmetic overflow in solidity," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498665

[17] Y. Feng, E. Torlak, and R. Bodik, "Summary-based symbolic evaluation for smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1141–1152. [Online]. Available: https://doi.org/10.1145/3324884.3416646

[18] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1678–1694.

[19] "The ethereum blockchain explorer," 2023, https://etherscan.io/.

[20] V. B. Fabian Vogelsteller, "A standard interface for tokens." 2015, https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[21] N. Reiff, "What crypto users need to know: The erc20 standard," 2021, https://www.investopedia.com/tech/why-crypto-users-need-know-about-erc20-token-standard/.

[22] O. Pomerantz, "Erc-20 contract walk-through," March 8, 2021, https://ethereum.org/en/developers/tutorials/erc20-annotated-code/.

[23] C. Team, "2022 biggest year ever for crypto hacking with $3.8 billion stolen, primarily from defi protocols and by north korea-linked attackers," 2023, https://blog.chainalysis.com/reports/2022-biggest-year-ever-for-crypto-hacking/.

[24] D. Siegel, "Understanding the dao attack," 2023, https://www.coindesk.com/learn/understanding-the-dao-attack/.

[25] T. Budd and F. Sayward, "Users guide to the pilot mutation system," *Yale University, New Haven, Connecticut, Technique Report*, vol. 114, 1977.

[26] T. A. Budd, R. J. Lipton, R. DeMillo, and F. Sayward, "The design of a prototype mutation system for program testing," in *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society, 1978, pp. 623–623.

[27] "Openzepplin's erc20 implementation," 2023, https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol.

[28] R. Just, "On effective and efficient mutation analysis for unit and integration testing," Ph.D., Ulm University, 2013.

[29] J. Offutt, "A mutation carol: Past, present and future," *Information & Software Technology*, vol. 53, pp. 1098–1107, 10 2011.

[30] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 268–279. [Online]. Available: https://doi.org/10.1145/351240.351266

[31] "Foundry forge documentation," 2023, https://book.getfoundry.sh.

[32] Certora and Openzepplin, "A library for secure smart contract development," 2022, https://github.com/Certora/openzeppelin-contracts.

[33] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009, source Code Analysis and Manipulation, SCAM 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584909000688

[34] M. Barboni, A. Morichetta, and A. Polini, "Sumo: A mutation testing strategy for solidity smart contracts," 2021.

[35] J. J. Honig, M. H. Everts, and M. Huisman, "Practical mutation testing for smart contracts," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, C. Pérez-Solà, G. Navarro-Arribas, A. Biryukov, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2019, pp. 289–303.

[36] A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang, "An extensible, regular-expression-based tool for multi-language mutant generation," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 25–28. [Online]. Available: https://doi.org/10.1145/3183440.3183485

[37] "Using gambit with the prover," 2023, https://docs.certora.com/en/latest/docs/gambit/mutation-verifier.html.

[38] Y. T. Chen, R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just, "Revisiting the relationship between fault detection, test adequacy criteria, and test set size," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 237–249. [Online]. Available: https://doi.org/10.1145/3324884.3416667

[39] T. Gündüzgiln, "Most common smart contract vulnerabilities (part 4) missing zero address validation," 2023, https://www.linkedin.com/pulse/most-common-smart-contract-vulnerabilities-part-4-zero-g%C3%BCnd%C3%BCzgil/.

[40] D. Amalfitano, A. C. R. Paiva, A. Inquel, L. Pinto, A. R. Fasolino, and R. Just, "How do java mutation tools differ?" *Communications of the ACM (CACM)*, pp. 1–23, Jan. 2022.

[41] S. Labs, "Awesome buggy erc20 tokens," 2023, https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/bad_tokens.top.csv.

[42] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, November 28–30 2012, pp. 11–20.

[43] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing*, vol. 7, 1997.

[44] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 1, p. 5–20, jan 1992. [Online]. Available: https://doi.org/10.1145/125489.125473

[45] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 433–436.

[46] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Practical mutation testing at scale: A view from google," *IEEE Transactions on Software Engineering (TSE)*, pp. 1–13, Aug. 2021.

[47] 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)*, 2018, pp. 47–53.

[48] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 275–378. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245818300305

[49] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[50] 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 *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, Hong Kong, Nov. 2014, pp. 654–665.

[51] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Inf. Process. Lett.*, vol. 60, no. 3, p. 135–141, nov 1996. [Online]. Available: https://doi.org/10.1016/S0020-0190(96)00135-4

[52] G. Mason and J. Pan, "Procedures for reducing the size of coverage-based test sets," 1995. [Online]. Available: https://api.semanticscholar.org/CorpusID:94732

[53] Z. Wei, W. Xiaoxue, Y. Xibing, C. Shichao, L. Wenxin, and L. Jun, "Test suite minimization with mutation testing-based many-objective evolutionary optimization," in *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*, 2017, pp. 30–36.

[54] R. Noemmer and R. Haas, "An evaluation of test suite minimization techniques," in *International Conference on Software Quality. Process Automation in Software Development*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:210510262

[55] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1398–1409. [Online]. Available: https://doi.org/10.1145/3368089.3417064

[56] 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*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 557–560. [Online]. Available: https://doi.org/10.1145/3395363.3404366

[57] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 322–333. [Online]. Available: https://doi.org/10.1145/3597926.3598059

[58] M. di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, 2019, pp. 69–78.