Gas limit aware mutation testing of smart contracts at scale

Pieter Hartel*†, Richard Schumi*
*Singapore University of Technology and Design, †Delft University of Technology

***Abstract*—The blockchain is a key technology that has been adopted in many application areas to increase security and reliability and to avoid the need for a central trusted authority. One of its essential underlying foundations are smart contracts, which are executable programs for managing data or assets on the blockchain. It is crucial that smart contracts are tested thoroughly due to their immutable nature and since even small bugs can lead to huge monetary losses. However, it is not enough to just test smart contracts, it is also important to ensure the quality and completeness of the tests. Hence, we introduce new smart contract specific mutation operators as well as a novel killing condition that is able to detect a deviation in the gas consumptions, i.e., in the monetary value that is required to perform transactions. Moreover, we establish a baseline for mutation testing of smart contracts by applying our method to a replay test suite and by testing about a thousand contracts.**

***Index Terms*—Mutation testing, Ethereum, Smart contracts, Gas limit as a killing criterion, Modifier issues.**

I. Introduction

Smart contracts are programs designed to express business logic for managing the data or assets on a blockchain system. They are becoming increasing popular with the rise of the blockchain in various disciplines. Although smart contracts already exist for some years, they still suffer from major security vulnerabilities, which can often lead to huge monetary losses. For example, it has been shown that simple coding mistakes can be exploited by attackers, which led to a loss of millions of dollars [1]. Hence, it is important to make sure that smart contracts do not contain such vulnerabilities. However, detecting known vulnerabilities as it was often done in the past [18] is only one aspect of checking the quality of smart contracts. It is also important to ensure they do not suffer from semantic errors that could result in unwanted behaviour, like a wrong calculation of a monetary value. The most important method for finding both vulnerabilities and semantic errors is testing. Testing smart contracts is even more important than testing regular programs, since their source is often publicly available, which makes them an easy target, and updating them is cumbersome due to their immutable nature. Moreover, it is critical to ensure the quality of the tests. There are a few simple quality metrics, like code coverage, i.e., the percentage of the source code that is executed by a test, but they are not able to measure the error detection capability of tests. A technique that can do this measurement is mutation testing, which injects faults into a program to check the ability of the tests to detect these faults.

In this paper, we discuss a mutation testing approach for Solidity[1] smart contracts that can check the ability of a test to reveal common smart contract faults. Solidity is a JavaScript like language with a number of special features to interact with the underlying Ethereum blockchain. The blockchain serves to store code and data, and the blockchain is managed by the owners of the Ethereum peer-to-peer network.

We introduce smart contract specific mutations that inject common mistakes that are made by smart contract developers. An example is a forgotten or wrong function modifier [12]. A modifier can express conditions that have to be fulfilled for the execution of a function, e.g., that the caller of the function is the owner of the contract. Since modifiers are often concerned with access control, it can have catastrophic effects when one is missing, e.g., important management functions of a smart contract can become publicly available. Hence, we introduce a mutation that replaces or deletes modifiers in order to check the ability of the tests to reveal this mistake.

Another smart contract specific aspect is the gas consumption of transactions. Everything on Ethereum costs some units of gas [34]. For example, executing an `ADD` bytecode costs 3 gas. Storing a byte costs 4 or 68 gas, depending on the value of the byte (zero or non-zero). The price of gas in Ether varies widely[2], and the market determines the exchange rate of Ether. The cost of a transaction can be anything from less than a cent to several US$. Executing smart contracts is therefore not just a matter of executing the code with the right semantics but also of cost control. Therefore, all transactions have a gas limit to make sure that the cost is managed. Executing smart contracts with a gas limit is comparable to executing code on a real time system with a deadline [22]. This opens up new possibilities for killing mutants, over and above the standard killing conditions.

Similar to detecting mutants on real-time systems with a different timing behaviour, we introduce a novel killing condition based on the gas consumption, i.e., we measure the gas consumption of tested transactions to find deviations to reference executions of these transactions. This allows us to kill mutants that consume a significantly greater amount of gas. Moreover, it can help to rule out some mutants that seem to have an equivalent functional behaviour, but that show different gas consumption.

Finally, we demonstrate the effectiveness of our method

[1] https://solidity.readthedocs.io/
[2] https://etherscan.io/chart/gasprice

at scale by checking the quality of replay tests [9] that are automatically produced from historic transaction data on the blockchain. The achieved mutation score can serve as a baseline for testing other, more sophisticated testing methods.

Our contributions are:

- We introduce novel mutation operators for Solidity, which are based on common programming mistakes, especially mistakes concerning missing access control.
- The gas limit of smart contract transactions is used as a novel killing condition. This improves the mutation score significantly.
- The approach is evaluated using a large sample of verified smart contracts downloaded from Etherscan with matching replay tests in order to establish a baseline for other testing methods.

II. Background

Mutation testing [15], [26], [27] is an evaluation technique for assessing the quality of a set of test cases (i.e., a test suite). It works by introducing faults into a system via source code mutation and by analysing the ability of the test suite to detect these faults. The idea is that the mutation should simulate common errors that a developer might make by mistake. Hence, when a test suite is able to find such artificial faults, it should also find real ones that can occur through programming mistakes. The quality of the test suite can be measured by the *mutation score*, which is calculated by dividing the number of detected (or killed) mutants by the total number of mutants. Ideally this score should be close to one.

Developers of smart contracts are likely to make some mistakes with standard language features, but because Ethereum is relatively young, they are more likely to confuse Solidity specific features. For example, Solidity offers two different types of assertions: `require(.)` is used to check external consistency, and `assert(.)` is used to check internal consistency. Both terminate the contract but with a different status that might confuse the distributed applications (DApp) that build upon the contract.

Developers also have trouble with the qualifiers that Solidity offers, such as `external` (for functions that can be called from other contracts and via transactions, but not internally), `public` (for functions that can either be called internally or via transactions), `internal` (for functions that can only be accessed internally), and `private` (for functions that are only visible in the contract they are defined in). There are more qualifiers and other subtle differences, which make it even harder to use the correct qualifier.

Finally, the addresses of contracts and externally owned accounts play such an important role in smart contracts that there are several ways of specifying addresses that may confuse the developer. For example, `address(this)` is the address of the current contract, which can also be accessed by a 40 hex digit address. `msg.sender` is the address of the sender of a message, and `tx.origin` is the address of the externally owned account that sent a transaction. They are the same for a short call chain but not for a longer call chain.

There are several papers that discuss typical mistakes of smart contract developers, including the security implications of confusing one notation for another [1]. We will not pursue this further and instead discuss a number of examples of mutations inspired by potential mistakes in subsection II-A.

To specify how faults should be injected, there are mutation operators [24] which are basically rules that describe the changes that should be made for a mutant. For example, a mutation operator might state that a constant integer should be replaced by another random integer.

Although mutation testing is an old technique, it has still open challenges, like the equivalent mutant problem, which occurs when a mutation does not change the original program, e.g., when a fault is injected in dead code. There are methods that try to detect or avoid equivalent mutants [8], [11], [23], [25], but it is still not possible to get rid of all of them. Hence, this limits the usability of mutation testing, since a high manual effort is required to identify equivalent mutants.

Every transaction on Ethereum costs gas. This creates an opportunity for mutation testing. For example, if two fragments of code are functionally equivalent but one costs more gas than the other, one could argue that they are not semantically equivalent. However, the tools that are currently available for estimating the amount of gas that a transaction needs are inaccurate [21]. Therefore, we suggest the question of how to use gas cost to determine functional equivalence of mutants as potential future work. We will focus here on leveraging gas consumption when deciding whether a mutant is killed.

Related approaches to our work have already applied mutation testing for domain specific languages or frameworks. For example, Zhu and Zaidman [37] presented a mutation analysis for the general purpose input output (GPIO) interface of IoT devices. Cañizares et al. [2] showed a mutation testing approach for a simulation framework for cloud and high performance computing environments.

There is also related work that introduces special mutation operators for specific languages features. For example, Le et al. [19] introduced dedicated mutation operators for functional programming idioms to enable an effective mutation analysis of functional programming languages. Similarly, there are also special mutation operators for WS-BPEL 2.0 [4].

Moreover, there are related approaches that introduced special mutation operators concerning non-functional properties, like energy usage or timing. Jabbarvand and Malek [14] introduced a mutation testing framework for Android that is able to detect bugs that result in a high energy consumption. They developed special energy-aware mutation operators that can create these energy bugs. A similar approach with mutation operators for time was presented by Nilsson et al. [22]. Their proposed method for real time systems applies mutations that change the timing behaviour of tasks and their aim was to produce tests that are able to identify these timing differences.

Both energy consumption and timing can be seen as related to the gas consumption of smart contract transactions. In the same way a mutant can be killed based on a run time or

energy deviation; it is also possible to kill a mutant based on a difference in the gas usage.

There are related papers that already introduced mutation testing for Solidity. Wu et al. [35] have implemented similar mutation operators and tested four DApps. The difference to our approach is that their mutations do not consider access control faults and they are not concerned with inspecting the gas usage. Honig et al. [12] do take access control into account, but not the gas limit. The evaluation of both papers is also limited to 4 respectively 2 contracts. Fu [5] et al. also introduced a smart contract mutation approach, but for testing implementations of the Ethereum Virtual Machine (EVM) implementations and not smart contracts. There are two GitHub repositories with related tools. Eth-mutants[3] implements just one mutation operator and UniversalMutator[4] describes a generic mutation tool [7] with set of operators for Solidity. These tools have not been evaluated.

The research question that follows from the analysis of the background is

> *How efficient are the standard mutation operators as compared to Solidity specific operators?*

To break this question down into its more manageable sub questions we present a case study in mutation testing of a sample smart contract first, and then list the sub questions.

A. A case study in mutation: Vitaluck

As a case study we use a lottery contract called Vitaluck [3]. The source of the contract can be browsed on Etherscan[5]. It contains a main method called `Play` and a number of management methods. The main method contains the core of the business logic of the lottery. Each call to `Play` draws a random number in the range 1 to 1000 using the time stamp of the current block as a source of entropy. If the random number is greater than 900, the player wins the jackpot. A percentage of each bet is paid to the owner of the contract.

Vitaluck is a relatively short contract (139 lines of source code excluding comments). It has not been used extensively; there are only 27 historic transactions that can also be browsed on Etherscan. The first transaction deploys the contract. The remaining historic transactions are all calls to the `Play` method. None of management methods of the contract are ever called by the historic transactions on the blockchain. However, the `Play` method is responsible for the majority of the code and provides ample opportunities for using standard mutation operators, as well as Solidity specific operators.

To motivate our choice for mutation operators we use a number of examples. Each example is labelled in italic font with the mutation operator and a brief description of the purpose of the operator. Moreover, we highlight some common known vulnerabilities, as described in the smart contract weakness classification (SWC) registry [31], which can be simulated with our mutation operators. Table II summarises the operators.

[3]https://github.com/federicobond/eth-mutants

[4]https://github.com/agroce/universalmutator

[5]https://etherscan.io/address/0xef7c7254c290df3d167182356255cdfd8d3b400b

AOR - Assignment Operator Replacement Since Vitaluck is a lottery, any mutation to the code that manages the jackpot has a high likelihood of causing a fault in the contract. The first sample mutation below (at line 123 of the Vitaluck source, using an output format inspired by the Unix `diff` command) requires the owner of the lottery to pay the player instead of the other way around. Such a fault is also described as a vulnerability by the SWC regisry: SWC-129 Typographical Error[6].

```
< currentJackpot = currentJackpot - MsgValue10Percent;
> currentJackpot = currentJackpot + MsgValue10Percent;
```

LR_I - Literal Integer replacement The second sample mutation (line 149) changes `900` to `1`. The range of `_finalRandomNumber` is 1 to 1000. If the condition in the `if` statement is true, the jackpot will be paid out, which in the original code happens on average 10% of the time. After the mutation, the jackpot will be paid out 99.9% of the time.

```
< if(_finalRandomNumber >= 900) {
> if(_finalRandomNumber >=  1 ) {
```

ITSCR - Identifier with same Type, Scope, and Constancy Replacement The third sample mutation (line 182) replaces the address of the winner of the jackpot by a hard coded address that belongs to the owner (the CFO) of the contract. This means that the jackpot will never be paid out to a player, but always to the CFO. Depending on which address (or identifier) is replaced, the operator can also introduce the vulnerability SWC-105 Unprotected Ether Withdrawal, e.g., when the name of the constructor is replaced, it will become publicly available. Introducing such vulnerabilities into a contract allows us to assess if a test suite is able to find them, and hence it can support the creation of thorough tests.

```
< currentWinningAddress.transfer(JackpotWon);
> cfoAddress.transfer(JackpotWon);
```

MORD - Modifier Replacement or Deletion* The fourth sample mutation (line 257) deletes the modifier `onlyCeo` from the method that installs the address of a new CEO. This allows anyone to set the pay-out address to his own, rather than just the CEO. This behaviour also corresponds to the vulnerability SWC-105, which can occur when the access control for functions is insufficient. The operator can further cause vulnerabilities, like the SWC-106 Unprotected SELFDESTRUCT Instruction, or SWC-123 Requirement Violation.

```
< function modifyCeo(address _newCeo) public onlyCeo {
> function modifyCeo(address _newCeo) public {
```

QRD - Qualifier for storage local or state mutability Replacement or Deletion* The fifth sample mutation (line 78) prevents anyone from playing the lottery, as removing the qualifier `payable` blocks any transfer of money to the contract. It can also change the accessibility of functions, which may reveal important initialization functions (SWC-100 Function Default Visibility).

[6]https://swcregistry.io/docs/SWC-129

```
< function Play() public payable {
> function Play() public {
```

LR_A - Literal Address replacement* The sixth sample mutation (line 30) changes the hard coded address of the CFO to the address of the contract itself. Assuming that CFO also controls the address of the contract, this should not be a problem for the lottery. However, it can also lead to a known authorization vulnerability as described by SWC-115, which can occur when the variable for the transaction origin *tx.origin* is used for authorization checks.

```
< address cfoAddress = 0xFFcf8FDEE72ac11b5c5...;
> address cfoAddress = address(this) ;
```

RAR - R-Value Address Replacement* The seventh sample mutation (line 191) prevents the contract from paying out winnings. The *RAR** operator replaces an R-Value (i.e., a value on the right hand side of an assignment operator). This can also lead to the same authorization vulnerability as for the LR_A* operator.

```
< currentWinningAddress = address(this);
> currentWinningAddress = address(0);
```

LR_B - Literal Boolean Replacement The final sample mutation (line 163) replaces the Boolean parameter of the `NewPlay` event.

```
< NewPlay(msg.sender, _finalRandomNumber, false);
> NewPlay(msg.sender, _finalRandomNumber, true );
```

This mutant returns incorrect information to the DApp. It has also been included because the changes to the amount of gas used during the deployment of the contract are relatively easy to explain. The cost of storing the bytecode of a contract depends not only on the size of the bytecode but also the values of the bytes to be stored. Storing a zero byte costs 4 gas units and storing a non-zero byte costs 68 gas. Since `false` is represented as 0 and `true` as a 1, we expect the amount of gas needed for deploying the mutant to increase by 64. Indeed, the original requires 1410422 gas, and the mutant requires 1410486 gas, which differ by 64. Since all other factors that cost gas involved in the deployment, such as the size of the code, and the bytecodes executed for deployment are the same for the mutant as for the original, the change of a 0 to a 1 byte accounts for the entire difference in gas usage between original and mutant. We can explore the deployment transaction of Vitaluck on the blockchain[7]. The gas limit turns out to be exactly enough to deploy the contract, i.e., 1410422 gas. With the same limit, the mutant above would run out of gas during deployment, and it should therefore be killed by the test.

The code structure and composition of a contract determines which opportunities there are for mutations. Vitaluck has 387 suitable program fragments, which can be broken down into statements, expressions etc. as shown in Table I. There are only 27 qualifiers (6%) and 4 modifier invocations. So most opportunities for mutations fit the standard operators, and few fit the Solidity specific operators.

[7]https://etherscan.io/tx/0x5f51cec3233e28959a2f39a7eec8bddc8178747e9a6b3991db0c02c6439fcc5b

TABLE I
MUTATION OPPORTUNITIES IN VITALUCK

#	Code fragment	#	Code fragment
28	Assignment	13	Member access
30	Binary operation	4	Modifier invocation
49	Expression statement	27	Qualifier
2	Function call	6	Return
153	Identifier	7	Unary operation
23	Literal	48	Variable declaration

B. Sub questions

Based on the Vitaluck case study we can now formulate a number of subsidiary research questions.

a) Discarding stillborn mutants: We created a simple tool (ContractMut) that makes maximum use of existing state-of-the-art tools, such as the Truffle framework[8] and the Solidity compiler. In particular, the tool relies heavily on the Solidity compiler to read the source file and to generate the abstract syntax tree (AST). This approach has the advantages that the amount of bespoke tooling to be built is limited. The disadvantage is that the compiler has much more information than it exposes via the AST. The mutation tool does not have semantic information about the original code or the mutant as it works on the AST generated by the Solidity compiler. This means that some mutants are generated that do not compile, for example:

- When replacing `*` by `-` in an expression such as `1 * 9 / 10000000000000000000`, an error occurs because a rational constant cannot be subtracted from an integer.
- If `+` mutates into `-` in an expression such as `new bytes(1+65)`, an error occurs because it is not possible to allocate a negative amount of storage.
- If in a declaration such as `a=b+1`, `b` is mutated to `a`, a cyclic dependency is created that draws an error.

This raises the sub question: *To what extent does the tool generate stillborn mutants?*

b) Discarding duplicate and equivalent mutants: Duplicate and equivalent mutants distort the mutation score. The next sub question is therefore: *How to detect duplicate and equivalent mutants?*

c) Mutation score: The purpose of generating mutants is to assess the quality of a test. If too few mutants are killed by a test, the usual course of action is to develop more tests. This is labour intensive. However, for a smart contract like Vitaluck that has already been deployed on the blockchain, all historic transactions are available and can be decompiled into a test. A test then consists of replaying the sequence of historic transactions. We can control the size of the test by varying the number of transactions executed by the mutant. As the size of the test increases, we expect the mutation score to increase. Unfortunately, most historic transactions execute the same method calls, so the increase in mutation score should

[8]https://www.trufflesuite.com

tail of rapidly. The sub question then is: *What is the relation between the mutation score and the length of the test?*

d) Efficiency of the mutation operators: We follow the competent programmer hypothesis [36] by creating a subtle variant of a program fragment that could have been created by a developer by mistake. The sub question is: *What is the relative success of standard mutation operators as compared Solidity specific operators?*

e) Using outputs in a mutant killing condition: To determine whether a mutant is killed, a test compares the output of the original to the output of the mutant and kills the mutant if there are differences. The next sub question is: *Which observable outputs can be used in the killing condition?*

f) Using gas in a mutant killing condition: The amount of gas used by a transaction depends on the number and type of the EVM instructions executed, the amount and type of data stored, and the amount of data sent to and received from the transaction [34]. Gas costs real money. Therefore, a prudent developer will try to limit the amount of gas available to a transaction. There are standard tools for this that use Worst Case Execution Time (WCET) estimation techniques [33]. The cautious developer uses these to estimate a limit bound on the amount of gas needed. A tight limit on the amount of gas reduces the risk of having to pay too much for a transaction. However, if the limit is too tight, some method calls may fail unexpectedly. Estimating the gas limit is a non-trivial problem. For example, even though EVM bytecodes are deterministic, a transaction may take inputs from a variety of sources, including the unpredictable time stamp of the current block. This makes it challenging to predict the gas limit. A prudent developer therefore might add at least a safety margin to ensure that a transaction will not run out of gas unexpectedly. This raises the following sub question: *To what extent is killing mutants based on exceeding the gas limit efficient?*

III. Method

This section describes an experiment in mutation testing of smart contracts on Ethereum. We have tried the simplest thing first to set a baseline. The experiments have been conducted on a uniform random sample of 1,120 smart contracts with tests that can be downloaded from Truffle-tests-for-free[9]. We removed 157 contracts from the set because either they were non-deterministic or they did not have a test with 50 transactions. These 963 contracts are representative for the entire collection of 50,000+ verified smart contracts on Etherscan [9], and the sample is relatively large [27]. The tests are replay tests and relatively short. The average bytecode coverage is 51.4% [9]. Our results are therefore a baseline.

A test for a contract begins by compiling and re-deploying the contract on a pristine blockchain. We use the Truffle framework for this with the exact same time stamps and transaction parameters as the historic deployment on the public Ethereum blockchain. All mainnet addresses are replaced by testnet addresses and all externally owned accounts have a generous balance. After re-deployment, the first 50 historic transactions are re-played, also with the historic time stamps and transaction parameters [9]. After each transaction all pure methods of a contract are called, with fuzzed parameters. This is intended to simulate any actions by a Distributed Application (DApp) built on top of the contract.

To detect whether a mutant is killed, we compare the outputs of a transaction generated by the original contract to the outputs of the corresponding transaction generated by the mutant contract. We compare only observable outputs of a transaction, which means that we consider only strong mutants that propagate faults to the outputs.

a) Discarding stillborn mutants: ContractMut uses the Solidity compiler to compile all the mutants it generates. If the compilation fails, the mutant is discarded. Smart contracts are usually relatively small, hence the time wasted on failed compilations is limited.

b) Discarding duplicate and equivalent mutants: ContractMut implements the trivially equivalent mutant detection method [17] to discarded duplicate and equivalent mutants. Each new mutant is compiled and the bytecode of the new mutant is compared to the bytecode of the original and the bytecode of all previously generated mutants. If there is a match, the new mutant is discarded.

c) Mutation score: Since the tests are machine generated, they are consistent in the sense that all sample contracts are tested by making 50 transactions. The advantage of using machine-generated tests is that this scales well to large numbers of contracts. The disadvantage is that the tests are not necessarily representative for handcrafted tests. For example, the bytecode coverage of the tests varies considerably, from 6% to 98% [9].

d) Efficiency of the mutation operators: ContractMut implements the essence of the Mothra set [16]. This is considered the minimum standard for mutation testing [27]. To this we added four Solidity specific mutation operators. Table II lists the operators in alphabetical order. The first column of the table indicates the correspondence with the Mothra operators. The operators marked with an asterisk are Solidity specific. The next two columns give the name of the operator, and a description. The next two columns give an example of use of the operator. The only operators that do not have a Mothra correspondence are the Solidity specific operators. Similarly, the Fortran specific operators from Mothra (DER, DSA, GLR) have not been implemented. We have also omitted the array operators (AARC, ACR, ASR, CAR, CNR, SAR) and absolute value insertion (ABS) because arrays are relatively rare and signed integers are extremely rare in smart contracts. The last column shows the relation of the operators to known vulnerabilities based on the smart contract weakness classification (SWC) registry [31].

Each node in the AST represents a program fragment that could in principle be mutated. Therefore, all relevant AST nodes are collected in a candidate list. This includes simple statements, literals, identifiers, function parameters, and

[9]https://github.com/pieterhartel/Truffle-tests-for-free

TABLE II
MUTATION OPERATORS FOR SOLIDITY PROGRAMS. OPERATORS MARKED WITH AN ASTERISK ARE SOLIDITY SPECIFIC.

Mothra operator	ContractMut operator	Description	Examples		SWC ID
			Replace this:	by this:	
AOR/LCE/ROR	AOR	Assignment Operator Replacement	`=`	`+=`	SWC-129
AOR/LCR/ROR	BOR	Binary Operator Replacement	`+`	`-`	SWC-129
			`&&`	`||`	SWC-129
SDL	ESD	Expression Statement Deletion	`a++`	`true`	
SVR	ITSCR	Identifier with same Type, Scope, and Constancy Replacement	`assert`	`require`	SWC-105, SWC-106
			`balance[t0]`	`balance[from]`	
			`emit Transfer(acc)`	`emit Approval(acc)`	
RSR	JSRD	Jump Statement Replacement or Deletion	`break`	`continue`	
-	LR_A*	Literal Address Replacement	`0x0xef7...b400b`	`msg.sender`	SWC-115
CRP/CSR/SCR/SRC	LR_B	Literal Boolean Replacement	`true`	`false`	
CRP/CSR/SCR/SRC	LR_I	Literal Integer Replacement	`10`	`11`	
CRP/CSR/SCR/SRC	LR_S	Literal String Replacement	`'ETH'`	`''`	
-	MORD*	Modified Replacement or Deletion	`onlyCeo`		SWC-105, SWC-106, SWC-123
-	QRD*	Qualifier for storage local or state mutability Replacement or Deletion	`public`	`private`	SWC-100, SWC-108
			`payable`		
			`bytes memory x`	`bytes storage x`	
-	RAR*	R-Value Address Replacement	`msg.sender`	`tx.origin`	SWC-115
AOR/LCR/ROR	UORD	Unary Operator Replacement or Deletion	`++`	`--`	SWC-129
SVR	VDTSCS	Variable Declaration with same Type Scope and Constancy Swap	`add(uint a,uint b)`	`add(uint b,uint a)`	

operators. Compound statements, methods and even entire contracts could also be included in the candidate list, but we assume that mutations to such larger program fragments are not consistent with the competent programmer hypothesis. Once the candidate list has been built, the tool repeatedly selects a mutation candidate uniformly at random from the list [20] and applies the appropriate mutation operator from Table II.

e) Using outputs in a mutant killing condition: To determine whether a mutant is killed the outputs of the original contract are compared to the outputs of the mutant while executing each transaction of the test. *TxEvMeth* compares all observable outputs of a transaction as follows:

- *Tx* compares the transaction status (i.e., success, failure, or out of gas).
- *Ev* compares all outputs of all events emitted by a transaction.
- *Meth* compares all outputs of all pure methods called by the DApp simulation after each transaction.

The combination *TxEvMeth* corresponds to the standard mutant killing condition.

f) Using gas in a mutant killing condition: Every historic transaction on the blockchain has a limit on the amount of gas used. To assess how tight these limits are, we have analysed the statistics of all $N = 50,930$ historic transactions downloaded from Truffle-tests-for-free. Columns two and three of Table III show that 15.9% of the sample uses exactly the gas limit, and 11.3% uses less than 20% of the gas limit. The minimum gas limit is 21,000 and the maximum is 8,000,029, which represents a large range. The standard deviation is also relatively large (1,002,691). Hence, there is considerable variance in how developers estimate the limit on transactions, and it is probably fair to say that the limits are not tight.

TABLE III
GAS USED BY THE HISTORIC TRANSACTION AS A PERCENTAGE OF THE HISTORIC GAS LIMIT VERSUS GAS USED BY THE REPLAYED TRANSACTION AS A PERCENTAGE OF THE CALCULATED GAS LIMIT.

	Historic		Replay	
% gas used of limit	(count)	(%)	(count)	(%)
$\geq 0\%$ & $< 20\%$	5062	11.3%	4292	12.3%
$\geq 20\%$ & $< 40\%$	6893	15.4%	5439	15.6%
$\geq 40\%$ & $< 60\%$	6405	14.3%	4314	12.4%
$\geq 60\%$ & $< 80\%$	9847	21.9%	7522	21.6%
$\geq 80\%$ & $< 100\%$	9564	21.3%	9119	26.2%
$= 100\%$	7127	15.9%	4171	12.0%
Tx success	44898	100.0%	34857	100.0%
Tx failed	6032		16073	
Total Tx	50930		50930	
Minimum gas limit	21000		21000	
Maximum gas limit	8000029		15279099	
Mean gas limit	397212		416350	
Std. deviation gas limit	1002691		1071615	

Let us assume that the limit as provided by a historic transaction, glh, is a hard limit on the amount of gas that the developer is prepared to pay. Then, in principle we can use this limit to kill all mutants executing the same transaction that exceed the limit. However, since we are replaying each historic transaction on the Truffle framework, the amount of gas used by replaying the transaction may be slightly different. To compensate for this, we propose to calculate the gas limit on replaying a transaction, glr, as the maximum of the gas limit of the historic transaction, and the scaled gas limit of the historic transaction. The scaling applied is the ratio of gur, the gas used by the replay, and guh, the gas used by the

historic transaction:

$$glr = max(glh, \frac{gur}{guh} glh)$$

The last two columns of Table III show the statistics of *glr* calculated according to the formula above. The distribution is similar to that of the gas used by the historic transaction, but there is more variance. The *max* operation in particular makes the limit on the gas used by the replay less tight than the gas limit on the historic transaction. In the next section, we will investigate to what extent *glr* is efficient as a killing condition. We will call this the *Limit* condition, and apply it on its own, and in combination with *TxEvMeth* as *TxEvMethLimit*.

IV. Results

This section describes the results of our experiment in mutation testing of smart contracts on Ethereum. For each smart contract with a test we tried to generate exactly 50 non-equivalent mutants. Since each attempt requires a call to the Solidity compiler, we set an upper limit of 1,000 on the number of attempts to generate a mutant. For 18 smart contracts fewer than 50 non-equivalent mutants were generated, but for the remaining 98.3% of the contracts we obtained 50 non-equivalent mutants. In total, we generated 71,314 mutants, of which 47,870 were non-equivalent. For each contract we then executed the tests against all non-equivalent mutants on the Truffle framework. We ran 47,870 $\times$ 50 = 2,393,500 transactions, which took over a week to run on 14 Linux virtual machines (Hardware: Intel Xeon dual core, 2.4 GHZ, with 16GB RAM).

a) Discarding stillborn mutants: Of the generated 71,314 mutants, 11,252 (15.8%) could not be compiled. As expected the *QRD** operator generates the most stillborn mutants: 63.3% of the mutants with this operator did not compile. We take this as an indication that using the limited amount of semantic information available in the Solidity AST is an acceptable approach towards building a baseline mutation tool. The percentage of stillborn mutants can be reduced to zero if the full power of various semantics analyses of the compiler could be leveraged, but the cost of building such a mutation tool just to reduce a relatively small percentage of failed compilations alone would not be justifiable.

b) Discarding duplicate and equivalent mutants: Of the 71,314 generated mutants, 12,192 (17.1%) were equivalent to the original or a duplicate of another mutant. The trivial equivalent detection method [17] that we used is therefore reasonably effective, especially since often about 40–45% of the mutants can be equivalent [8], [30].

c) Mutation score: Figure 1 shows how the mutation score increases with the test size. The error bars for a 95% confidence interval are small. The standard mutant killing condition *TxEvMeth* has most success early on, whereas the success of the *Limit* condition increases more gradually. This difference can be explained as follows. All tests execute the constructor method in transaction 0 and one regular method in transaction 1. A relatively large fraction of the tests only execute these two methods (see below for details), hence most of the opportunity for killing a mutant on regular outputs occurs during transactions 0 and 1.

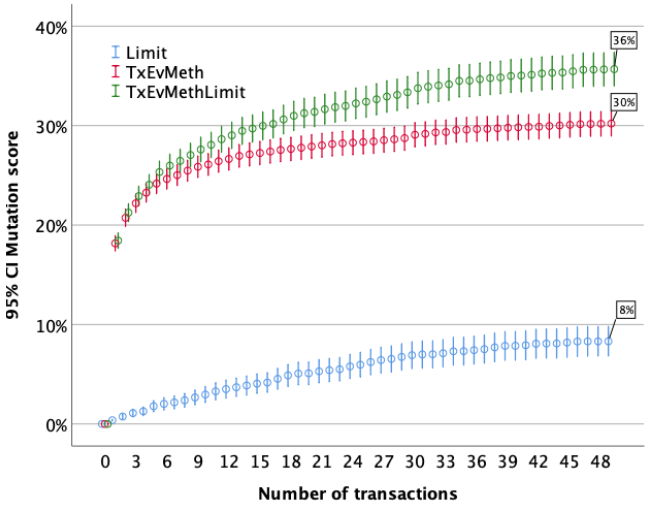

Fig. 1. Percentage of non-equivalent mutants killed as a function of the length of the test. The error bars correspond to a confidence interval of 95%.

Since in general size matters [13], we fixed the size of the tests to 50 transactions. However, we could not fix the size of the smart contracts. To study the influence of contract size we have calculated the rank correlation of the size of the bytecode and the mutation score. For the combined killing condition *TxEvMethLimit*, we found Kendall's $\tau = -0.11$ ($p = 0.01$, 2-tailed). This means that the mutation score is not correlated with the size of the test. We also calculated the correlation between the mutation score and the fraction of bytecodes that were executed by the test and found the correlation to be moderate: Kendall's $\tau = 0.45$ ($p = 0.01$, 2-tailed). The literature reports similar figures [6].

It is illustrative to compare our mutation scores with some of the results from the literature. Javalanche [29] achieved 5% – 63% for a set of seven Java programs of 2K – 95K lines of code (LoC), and Tengeri et al. [32] report raw mutation scores of 32% – 84% for a set of four Java programs of 31K – 229K LoC. Rojas et al. [28] performed a mutation study with students that tested 20 Java classes with 102 – 261 LoC, which achieved mutation scores of about 69%, and they also applied automated random testing tools which achieved scores of 26% – 45%. It can be seen that our achieved mutation scores are comparable to these studies. Mutation approaches with manual tests sometimes achieve higher scores, but they often require many iterations to improve the tests, which was not feasible in our case. Our approach has the advantage that it works with a high number of existing contracts, which enables a rigorous analysis of our mutation operators. Moreover, we did not find any literature with a mutation analysis of replay tests that also reports their scores and evaluates a reasonable set of test programs. So there seems to be no equal work that allows a fair comparison.

d) Efficiency of the mutation operators: Table IV shows to what extent the mutation operators from Table II have been successful. The first column gives the name of the mutation operator. The second column indicates how many mutants were not killed. The third column gives the number of mutants that were killed. The fourth column gives the total number of non-equivalent mutants. The last two columns give the percentages related to the numbers in columns two and three.

TABLE IV
CONTINGENCY TABLE OF THE MUTATION OPERATORS AGAINST THE MUTATION SCORE WITH THE *TxEvMethLimit* MUTANT KILLING CONDITION.

Mutation	Non-equivalent mutants			Percentage	
operator	Not killed	Killed	Total	Not killed	Killed
AOR	2178	1039	3217	67.7%	32.3%
BOR	3549	2245	5794	61.3%	38.7%
ESD	5246	2595	7841	66.9%	33.1%
ITSCR	7796	4085	11881	65.6%	34.4%
JSRD	1866	1090	2956	63.1%	36.9%
LR_A*	46	58	104	44.2%	55.8%
LR_B	973	225	1198	81.2%	18.8%
LR_I	1526	1089	2615	58.4%	41.6%
LR_S	158	329	487	32.4%	67.6%
MORD*	921	143	1064	86.6%	13.4%
QRD*	1041	1761	2802	37.2%	**62.8%**
RAR*	3002	1158	4160	72.2%	27.8%
UORD	297	189	486	61.1%	38.9%
VDTSCS	2215	1050	3265	67.8%	32.2%
Total	30814	17056	47870	64.4%	35.6%

$\chi^2 = 1759.6, df = 13, p < 0.001$

Contingency Table IV shows that of the four Solidity specific operators (marked with an asterisk) *QRD** is the most efficient. This is because subtle changes to the qualifiers, such as removing the `payable` attribute from a method completely breaks the contract. The standard *LR_S* operator is the most efficient operator overall. The reason is that strings in Solidity are typically used for communication with the DApp built on top of a smart contract. This means that even the smallest change to a string will be detected by comparing event parameters or method results. The *MORD** operator has the lowest efficiency (13.4%). This is because relatively few historic transactions try to violate the access control implemented by the modifiers. The *RAR** operator also has a low efficiency, for the same reason: few historic transaction try to exploit bugs in address checking.

e) Using outputs in a mutant killing condition: The mutation score for *TxEvMethLimit* reaches 35.7%, for *TxEvMeth* 30.2%, and for *Limit* 8.3%. These mutation scores are low compared to state of the art approaches [27]. However, this figure is useful as a baseline for other approaches that use realistic tests instead of replay tests, and our aim was not to achieve a high mutation score, but to evaluate the applicability of our operators for a high number of contracts. Moreover, even such a small score is helpful to show what kind of tests are missing and what has to be done to improve the tests.

f) Using gas in a mutant killing condition: The contribution of the gas limit as killing condition is 5.5%, which seems rather small, but since it can require a huge manual effort to analyse the surviving mutant, even such a small fraction can save hours or days of manual work.

V. SUMMARY AND DISCUSSION

We put the results in a broader context and formulate the answers to the research question.

a) Discarding stillborn mutants: We believe that the percentage of the generated mutants that do not compile was relatively small and that it was acceptable in the exploratory context. A production tool would have to implement more sophisticated mutations and would therefore have more knowledge of the semantics of smart contracts.

b) Discarding duplicate and equivalent mutants: A simple but state of the art approach has been used to address the equivalent mutant problem. What we have not investigated is to what extent gas usage can be leveraged to discard more equivalent mutants. We suggest this as a topic for future work.

c) Mutation score: By leveraging the historic data available on the Ethereum blockchain we have been able to generate tests that can be truncated to explore the relationship between test strength and the mutation score.

d) Efficiency of the mutation operators: One of the Solidity specific mutation operators was found to be more effective in producing mutants that have a high chance of being killed than the standard mutation operators. We take this as an indication that further research is needed to develop more sophisticated Solidity specific mutation operators. Another, important aspect of mutation operators is to which extent they can introduce common or severe bugs. To assess this quality, we inspected the ability of our operators to introduce known vulnerabilities which can have severe consequence. The associated vulnerabilities as (specified by the SWC) are shown in Table II. It can be seen that our specific operators are concerned with more vulnerabilities and that they are more severe compared to the standard operators. For example, *MORD** can trigger three different kinds of access control related vulnerabilities, which would not be possible with the standard operators.

e) Using outputs in a mutant killing condition: Comparing the observable outputs of a transaction between the mutant and the original kills most mutants.

f) Using gas in a mutant killing condition: We have shown that using the gas consumption as a killing condition can improve the mutation score and hence the effectiveness of the mutation approach in general.

We are now able to answer the main research question: Our Solidity specific mutation operators are more efficient than the standard operators, and they also are able to introduce more and more severe vulnerabilities. Table V shows that the difference in efficiency is 3.3 percent point, which is modest, but also statistically significant at ($p < 0.001$).

TABLE V
COMPARISON OF ALL MUTANTS VERSUS THE MANUALLY ANALYSED MUTANTS.

Mutation operator	All mutants Killed	Total	Manually analysed Killed	Total
Mothra	35.1%	39740	41.3%	223
Solidity	38.4%	8130	48.1%	27
Total	35.6%	47870	42.0%	250

$\chi^2 = 32.2, df = 1, p < 0.001$

VI. LIMITATIONS AND THREATS TO VALIDITY

We first discuss some general threats to the validity of our method and then some of its limitations based on the recently published seven point checklist for mutation testing in controlled experiments [27].

A. Threads to Validity

A threat to the validity of our evaluation might be that we only consider a replay test suite that is less powerful than other testing techniques, which might obtain a higher mutation score. Although there are better testing techniques, the focus of this work was not to find them, but to build a mutation-based test quality assurance method that can also serve as a baseline for other testing techniques of smart contracts.

To assess the effect of using a replay test suite we have analysed by hand all 250 mutants generated for 5 carefully selected smart contracts. Each contract took us about one day to analyse, hence we had to limit the number of contracts to a small number like 5. We used a stratified sampling method taking one contract for the top 5% contracts by mutation score, one contract from the bottom 5%, and one contract each from $25 \pm 2.5\%$, $50 \pm 2.5\%$ and $75 \pm 2.5\%$. In selecting the contracts from the five ranges, we tried to avoid analysing the same type of contract more than once. We analysed: the token DBToken[10], the wallet MultiSigWallet[11], the auction NumberBoard[12], the game casinoProxy[13], and the asset manager mall[14].

Table V shows that the key statistics of the 250 manually analysed mutants and all 47,870 mutants are comparable, which suggests that the results we obtain for the 250 mutants are representative for all mutants.

Table VI shows a contingency table for the mutation operator types versus the status of the different mutation results. The Mohtra operators generate more semantically equivalent mutants than the solidity operators, but overall the percentage of semantically equivalent mutants is only 3.2%. This indicates that semantically equivalent mutants do not inflate the base line statistics by more than a few per cent points. The second conclusion that can be drawn from Table VI is that almost 100% mutation scores are possible with better tests. A mutation score of 100% is not achievable with the current implementation because it cannot detect the success or failure of an internal transaction.

The *MORD** operator helped us to discover a vulnerability in one of the contracts, which we have reported to the owners by way of responsible disclosure. The vulnerability consists of a missing modifier `onlyOwner` on one of the methods.

TABLE VI
CONTINGENCY TABLE OF THE MUTATION OPERATOR TYPES AGAINST THE RESULTS OF MANUALLY ANALYSING 250 MUTANTS.

Mutation operator	Not killed	Semantically equivalent	Killed or killable	Total (count)
Mothra	0.0%	3.1%	96.9%	223
Solidity	3.7%	3.7%	92.6%	27
Total	0.4%	3.2%	96.4%	250

$\chi^2 = 8.3, df = 2, p = 0.016$

Another argument regarding the validity of our method might be that it is not wise to kill a mutant only based on a different gas usage since it could still be semantically equal. However, we believe that a different gas usage is still a valid reason to kill a mutant, because it represents a change in the monetary cost of a transaction. Moreover, there are other comparable cost factors, like energy consumption [14] or execution time [22] that have been used as killing condition in the past.

B. Mutation Testing Checklist

a) Mutant selection: The mutation operators consist of the minimum standard Mothra set plus a number of experimental Solidity specific operators. We have provided examples to motivate the choice of operators in subsection II-A, but this does not constitute proof that common errors in Solidity programs are captured by exactly the chosen operators. We leave such a proof for future work.

b) Mutation testing tool: We have implemented a bespoke mutation tool in JavaScript and the source will be made available on GitHub. The appendix provides a detailed specification of the mutation operators. The mutation generation tool uses uniform random selection of mutation operators and operands.

Calls to `block.timestamp` do not cause non-deterministic behaviour as all transactions in the tests happen in the test environment at exactly the same time as they happened historically. Calls to `block.blockhash` may generate non-deterministic results, but we excluded contracts that make such calls.

c) Mutant redundancy: We have used the state-of-the-art method [17] to discard redundant mutants.

d) Test suite choice and size: We have used a relatively large sample of smart contracts that are representative for the entire collection of verified smart contracts available from Etherscan. This collection however, is probably not representative for the entire population of smart contracts. For example, many smart contracts in this larger population are clones of

[10] https://etherscan.io/address/0x42a952Ac23d020610355Cf425d0dfa58295287BE
[11] https://etherscan.io/address/0xa723606e907bf84215d5785ea7f6cd93a0fbd121
[12] https://etherscan.io/address/0x9249133819102b2ed31680468c8c67F6Fe9E7505
[13] https://etherscan.io/address/0x23a3db04432123ccdf6ef4459684329cc7c0b022
[14] https://etherscan.io/address/0x3304a44aa16ec40fb53a5b8f086f230c237f683d

other smart contracts [10], but this is not the case for verified smart contracts from Etherscan. From a uniform random sample of 500 verified smart contract we have calculated the normalised edit distance (NLD) of all pairs and found that less than 1% of the pairs have a NLD of less than 20%, and less than 5% of all pairs have a NLD of less than 60%. This means that the vast majority of all pairs are different.

The tests are all of the same size, as each makes exactly 50 calls to a contract method. The tests have not been designed as tests by developers of smart contracts, and can therefore only be regarded as a baseline for real tests.

e) Clean program assumption: A limitation of this study is that we rely implicitly on the clean program assumption because we use existing smart contracts.

f) Multiple experimental repetitions: We have performed the main experiment with a relatively large number of smart contracts. In a second experiment, we have tried to improve the test coverage by downloading double the number of historic transactions (i.e., 100) for a uniform sub sample of 63 smart contracts, but the results remain essentially the same.

g) Presentation of the results: The results are presented aggregated over the entire sample of smart contracts, which is larger than in most related research.

VII. Conclusions and future work

Smart contracts are programmed in domain specific languages such as Solidity. The learning curve is steep, and the technology is young, so that developers are relatively inexperienced. Therefore, it is important that at least the tooling is up to date. We have proposed a mutation tool for Solidity that developers can use to improve their tests.

We have evaluated the mutation tool on a large and realistic random sample of smart contracts downloaded from Etherscan. The tests used in the evaluation replay historic transactions downloaded from Truffle-tests-for-free. Similar transactions typically form the core of real handcrafted tests for smart contracts, but these would be more comprehensive. Replay tests form a baseline for mutation testing of smart contracts.

We propose a set of mutation operators for Solidity, which includes the standard mutation operators from the Mothra set, as well as four specific operators for Solidity. We showed that our specific operators are on average a bit more efficient than the standard Mothra operators. They are also able to introduce more and more severe vulnerabilities, which is important to assess the detection capabilities of the tests.

This shows that bespoke mutation operators are useful. The average mutation score that we reported are not as good as other state-of-the-art mutation tools because replay tests have a limited bytecode coverage. Using the gas limit as a killing condition has improved the mutation score by up to 5.5 per cent points.

By way of future work, it would be expedient to repeat the experiments with more realistic tests created by developers. It would also be useful to study errors made by Solidity developers at scale to validate the mutation operators. Another area of future work would be to use the gas limit on transactions to detect equivalent mutants.

Acknowledgments

This work was supported in part by the National Research Foundation (NRF), Prime Minister's Office, Singapore, under its National Cybersecurity R&D Programme (Award No. NRF2016NCR-NCR002-028) and administered by the National Cybersecurity R&D Directorate.

We thank Sun Jun and Maarten Everts for comments on a draft of the paper.

Appendix - Mutation operators

This appendix lists all mutation operators as sets of pairs. The left component of the pair is replaced uniformly at random by (one of) the right component of the pair.

AOR: {(/=,%=), (%=,/=), (=,{+=,-=,*=}), (+=,{=,-=,*=}), (-=,{=,+=,*=}), (*=,{=,+=,-=}), (>>=,<<=), (<<=,>>=), (|=,{^=,&=}), (^=,{|=,&=}), (&=,{|=,^=})}

BOR: {(&&,||), (||,&&), (+,{-,*}), (-,{+,*}), (*,{+,-}), (/,%), (%,/), (<,{>,<=,>=,==,!=}), (>,{<,<=,>=,==,!=}), (<=,{<,>,>=,==,!=}), (>=,{<,>,<=,==,!=}), (==,{<,>,<=,>=,!=}), (!=,{<,>,<=,>=,==}), (>>,<<), (<<,>>), (|,{^,&}), (^,{|,&}), (&,{|,^})}

ESD: {(expression_statement,`true`)}

ITSCR: {(identifier_1,identifier_2), (identifier_2,identifier_1)}

JSRD: {(`break`,{`continue`,`true`}), (`continue`,{`break`,`true`}), (`return`,{`throw`,`true`}), (`throw`,{`return`,`true`})}

LR_A*: {(`0x`...,{`address(.)`,`msg.sender`,`tx.origin`,`address(0)`, `address(this)`}), (`address(.)`,{`msg.sender`,`tx.origin`, `address(0)`,`address(this)`})}

LR_B: {(`true`,`false`), (`false`,`true`)}

LR_I: {(number,{number+1,number-1,0,1})}

LR_S: {('string','')}

MORD*: {(modifier_1,{modifier_2,␣})}

QRD*: {(`memory`,{`storage`,`calldata`,␣}), (`storage`,{`memory`,`calldata`,␣}), (`calldata`,{`memory`,`storage`,␣}), (`public`,{`private`,`internal`,`external`,`constant`,␣}), (`private`,{`public`,`internal`,`external`,`constant`,␣}), (`internal`,{`public`,`private`,`external`,`constant`,␣}), (`external`,{`public`,`private`,`internal`,`constant`,␣}), (`constant`,{`public`,`private`,`internal`,`external`,␣}), (`payable`,␣)}

RAR*: {(`msg.sender`,{`tx.origin`,`address(0)`,`address(this)`}), (`tx.origin`,{`msg.sender`,`address(0)`,`address(this)`}), (`address(0)`,{`msg.sender`,`tx.origin`, `address(this)`}), (`address(this)`,{`msg.sender`,`tx.origin`,`address(0)`})}

UORD: {(!,␣), (++,{--,␣}), (--,{++,␣}), (-,␣), (~,␣)}

VDTSCS: {(declaration_1,declaration_2), (declaration_2,declaration_1)}

References

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In Matteo Maffei and Mark Ryan, editors, *6th Conf. on Principles of Security and Trust (POST)*, volume 10204 of *LNCS*, pages 164–186, Uppsala, Sweden, Apr 2017. Springer. URL: https://doi.org/10.1007/978-3-662-54455-6_8.

[2] Pablo C. Cañizares, Alberto Núñez, and Mercedes G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018. URL: https://doi.org/10.1016/j.jss.2018.05.010.

[3] Vincent Chia, Pieter Hartel, Qingze Hum, Sebastian Ma, Georgios Piliouras, Daniël Reijsbergen, Mark van Staalduinen, and Pawel Szalachowski. Rethinking blockchain security: Position paper. In Mohammed Atiquzzaman, Jin Li, and Weizhi Meng, editors, *Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics*, pages 1273–1280, Halifax, Canada, Jul 2018. IEEE. URL: https://doi.org/10.1109/Cybermatics_2018.2018.00222.

[4] Antonia Estero-Botaro, Francisco Palomo-Lozano, and Inmaculada Medina-Bulo. Quantitative evaluation of mutation operators for ws-bpel compositions. In *3rd Int. Conf. on Software Testing, Verification, and Validation (ICSTVV)*, pages 142–150, Paris, France, Apr 2010. IEEE. URL: https://doi.org/10.1109/ICSTW.2010.36.

[5] Ying Fu, Meng Ren, Fuchen Ma, Yu Jiang, Heyuan Shi, and Jiaguang Sun. Evmfuzz: Differential fuzz testing of Ethereum virtual machine. Technical report, Tsinghua University, China, Apr 2019. URL: https://arxiv.org/abs/1903.08483.

[6] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *36th Int. Conf. on Software Engineering (ICSE)*, pages 72–82, Hyderabad, India, May 2014. ACM. URL: http://doi.org/10.1145/2568225.2568278.

[7] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *40th Int. Conf. on Software Engineering: Companion Proceeedings (ICSE)*, pages May 25–28, Gothenburg, Sweden, 2018. ACM. URL: http://doi.acm.org/10.1145/3183440.3183485.

[8] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*, pages 192–199. IEEE Computer Society, 2009. URL: https://doi.org/10.1109/ICSTW.2009.37.

[9] Pieter Hartel and Mark van Staalduinen. Truffle tests for free – replaying Ethereum smart contracts for transparency. Technical report, Singapore Univ. of Technology and Design, Singapore, Jul 2019. URL: https://arxiv.org/abs/1907.09208.

[10] Ningyu He, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang. Characterizing code clones in the Ethereum smart contract ecosystem. Technical report, Beijing University of Posts and Telecommunications, May 2019. URL: https://arxiv.org/abs/1905.00272.

[11] Robert M. Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Softw. Test., Verif. Reliab.*, 9(4):233–262, 1999. URL: https://doi.org/10.1002/(SICI)1099-1689(199912)9:4⟨233::AID-STVR191⟩3.0.CO;2-3.

[12] Joran J. Honig, Maarten H. Everts, and Marieke Huisman. Practical mutation testing for smart contracts. In *Int. Workshop on Cryptocurrencies and Blockchain Technology (CBT)*, pages 289–303, Luxembourg, Sep 2019. Springer. URL: https://doi.org/10.1007/978-3-030-31500-9_19.

[13] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *36th Int. Conf. on Software Engineering (ICSE)*, pages 435–445, Hyderabad, India, 2014. ACM. URL: http://doi.org/10.1145/2568225.2568271.

[14] Reyhaneh Jabbarvand and Sam Malek. μdroid: an energy-aware mutation testing framework for android. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 208–219. ACM, 2017. URL: https://doi.org/10.1145/3106237.3106244.

[15] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on software engineering*, 37(5):649–678, Sep 2011. URL: https://doi.org/10.1109/TSE.2010.62.

[16] Kim N. King and A. Jefferson Offutt. A fortran language system for mutation-ased software testing. *Software-Practice and Experience*, 21(7):685–718, Jul 1991. URL: https://doi.org/10.1002/spe.4380210704.

[17] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on software engineering*, 44(4):308–333, Apr 2018. URL: https://doi.org/10.1109/TSE.2017.2684805.

[18] Denis Kirillov, Oleg Iakushkin, Vladimir Korkhov, and Vadim Petrunin. Evaluation of tools for analyzing smart contracts in distributed ledger technologies. In *Computational Science and Its Applications (ICCSA)*, pages 522–536, Saint Petersburg, Russia,, Jul 2019. Springer International Publishing. URL: https://doi.org/10.1007/978-3-030-24296-1_41.

[19] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mutation testing of functional programming languages. Technical report, Technical report, Oregon State University, 2014.

[20] Lu Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *32nd Int. Conf. on Software Engineering (ICSE)*, pages 435–444, Cape Town, South Africa, May 2010. ACM. URL: http://doi.org/10.1145/1806799.1806863.

[21] Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha" Sharygina. Computing exact worst-case gas consumption for smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice (IsoLa)*, volume 11247 of *LNCS*, pages 450–465, Limassol, Cyprus, Nov 2018. Springer. URL: https://doi.org/10.1007/978-3-030-03427-6_33.

[22] Robert Nilsson, Jeff Offutt, and Jonas Mellin. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer Science*, 164(4):97–114, Oct 2006. URL: https://doi.org/10.1016/j.entcs.2006.10.010.

[23] A. Jefferson Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Softw. Test., Verif. Reliab.*, 4(3):131–154, 1994. URL: https://doi.org/10.1002/stvr.4370040303.

[24] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996. URL: https://doi.org/10.1145/227607.227610.

[25] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test., Verif. Reliab.*, 7(3):165–192, 1997. URL: https://doi.org/10.1002/(SICI)1099-1689(199709)7:3⟨165::AID-STVR143⟩3.0.CO;2-U.

[26] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: Uniting the orthogonal. In Eric W. Wong, editor, *Mutation Testing for the New Century*, pages 34–44. Springer, Boston MA, 2001. URL: https://doi.org/10.1007/978-1-4757-5939-6_7.

[27] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier Inc., 2019. URL: https://doi.org/10.1016/bs.adcom.2018.03.015.

[28] José Miguel Rojas, Thomas D. White, Benjamin S. Clegg, and Gordon Fraser. Code defenders: crowdsourcing effective tests and subtle mutants with a mutation testing game. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 677–688. IEEE / ACM, 2017. URL: https://doi.org/10.1109/ICSE.2017.68, doi:10.1109/ICSE.2017.68.

[29] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for Java. In *7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, pages 297–298, Amsterdam, The Netherlands, Aug 2009. ACM. URL: https://doi.org/10.1145/1595696.1595750.

[30] David Schuler and Andreas Zeller. (un-)covering equivalent mutants. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 45–54. IEEE Computer Society, 2010. URL: https://doi.org/10.1109/ICST.2010.30.

[31] SmartContractSecurity. Smart contract weakness classification registry. https://github.com/SmartContractSecurity/SWC-registry/, 2019.

[32] Dávid Tengeri, Lászlo Vidács, Árpád Beszédes, Judit Jász, Gergö Balogh, Béla Vancsics, and Tibor Gyimóthy. Relating code coverage, mutation score and test suite reducibility to defect density. In *9th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 174–197, Xian, China, Apr 2016. IEEE. URL: https://doi.org/10.1109/ICSTW.2016.25.

[33] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, Apr 2008. URL: https://doi.org/10.1145/1347375.1347389.

[34] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger - eip-150 revision. Technical report 759dccd, Ethcore.io, Aug 2017. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[35] Haoran Wu, Xingya Wang, Jiehui Xu, Weiqin Zou, Lingming Zhang, and Zhenyu Chen. Mutation testing for Ethereum smart contract. Technical report, Nanjing University, China, Aug 2019. URL: https://arxiv.org/abs/1908.03707.

[36] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. A systematic literature review of how mutation testing supports quality assurance processes. *J. of Software: Testing verification and reliability*, 28(6):e1675:1–39, Sep 2018. URL: https://doi.org/10.1002/stvr.1675.

[37] Qianqian Zhu and Andy Zaidman. Mutation testing for physical computing. In *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*, pages 289–300. IEEE, 2018. URL: https://doi.org/10.1109/QRS.2018.00042.