Practical Mutation Testing for Smart Contracts

Joran J. Honig[1,3](✉), Maarten H. Everts[1,2], and Marieke Huisman[1]

[1] University of Twente, Enschede, The Netherlands
{maarten.everts,m.huisman}@utwente.nl
[2] TNO, The Hague, The Netherlands
[3] ConsenSys, New York, USA
joran.honig@consensys.net

Abstract. Solidity smart contracts operate in a hostile environment, which introduces the need for the adequate application of testing techniques to ensure mitigation of the risk of a security incident. Mutation testing is one such technique. It allows for the evaluation of the efficiency of a test suite in detecting faults in a program, allowing developers to both assess and improve the quality of their test suites. In this paper, we propose a mutation testing framework and implement a prototype implementation called Vertigo that targets Solidity contracts for the Ethereum blockchain. We also show that mutation testing can be used to assess the test suites of real-world projects.

Keywords: Mutation testing · Smart contract · Solidity

1 Introduction

Recent developments in distributed computing have resulted in platforms that support the execution of so-called "smart contracts". In this paper, we will look specifically at smart contracts written in the Solidity programming language [12], for the Ethereum blockchain. These smart contracts are programs that are executed by the nodes in the Ethereum network and can deal with votes, money transfers and even digital collectibles, such as Cryptokitties [3]. Because smart contracts exist on the Ethereum blockchain, they are openly readable and executable by any participant in the network. Furthermore, once a contract is uploaded it cannot be changed anymore, it is immutable. These properties force smart-contract developers to be very conscientious about the correctness and security of their code.

To reason about the correctness and security of smart contracts, developers employ different testing techniques. One of these techniques is unit testing, a commonly accepted practice in traditional software development. It has clear benefits and allows teams around the world to develop and release their software with confidence.

However, it is often very difficult to provide adequate guarantees over the correctness and security of a system using unit tests. As a result, critical vulnerabilities have frequently been discovered in live contracts. Notable security incidents include the hack of “The DAO” [14], where an attacker was able to exploit a re-entrancy vulnerability, allowing the attacker to withdraw 60 million USD worth of Ether, the base currency of Ethereum. Another example is the Parity wallet bug [8], where a user accidentally self-destructed code on which many other contracts were dependent, freezing assets worth 280 million USD at the time.

One factor that makes the application of unit testing difficult to do well is accurately described by Dijkstra, who once wrote, “Program testing can be used to show the presence of bugs, but never to show their absence!” [19]. This paints a grim picture, as it indicates that even if there are a multitude of tests, then there is still no guarantee that there are no bugs in the program. However, it also emphasises what unit tests can be good at, namely showing the presence of bugs. One needs to be able to gauge the effectiveness of a test suite at detecting bugs, to effectively apply unit testing to smart contracts. A good metric can give insight into both the guarantees given by a test suite and possible areas of improvement. One metric that is frequently used is code coverage, which gives the developer a concrete idea of which parts of the code are covered. However, code coverage is flawed and does not give a direct indication of the test suite’s effectiveness for detecting bugs. An example of a flawed test suite, which scores well with this metric is one without assertion statements, which could cover all the lines of code, but would not give any guarantees whatsoever.

Mutation testing [23] is a technique which aims to mitigate this problem. Instead of using the amount of code covered as a heuristic for the quality of the test suite, it measures the suite’s effectiveness at detecting faults in the source code directly. It does this by procedurally introducing small mutations in the source code, for example, a change from “>” to “<=”, and executing the test suite for each such mutation. After having tested all the mutations, the ratio between the detected mutations (also called “killed” mutations), and the undetected mutations (also called “surviving mutations”) can be used as a metric to evaluate the quality of the test suite.

This metric can provide valuable input in multiple stages of the software development lifecycle. A threshold value for the mutation score can be used to set a quality gate before a software release. Additionally, the information on mutants that have survived (i.e. have not been killed) can be used to increase the quality of the test suite, as these mutants identify specific edge cases which are not yet covered by the current test suite. An overview of the surviving mutations is also useful for a security review, as these mutants indicate locations where the program behaviour might not match the expectation of the developer.

This paper introduces Vertigo, a mutation testing tool written for Truffle and Solidity. We will describe the design choices, limitations, and further work for this implementation. Additionally we provide and discuss the application of mutation testing in the smart contract development lifecycle.

The two main contributions of this paper are:

– We provide a detailed design and implementation of a mutation testing framework agnostic of its test environment. The implementation is designed to be extensible to facilitate the implementation of future research.
– We propose new mutation operators that can be applied to Solidity.

2 Background

This section provides an overview of the current state-of-the-art for mutation testing and Ethereum smart contracts.

2.1 Mutation Testing

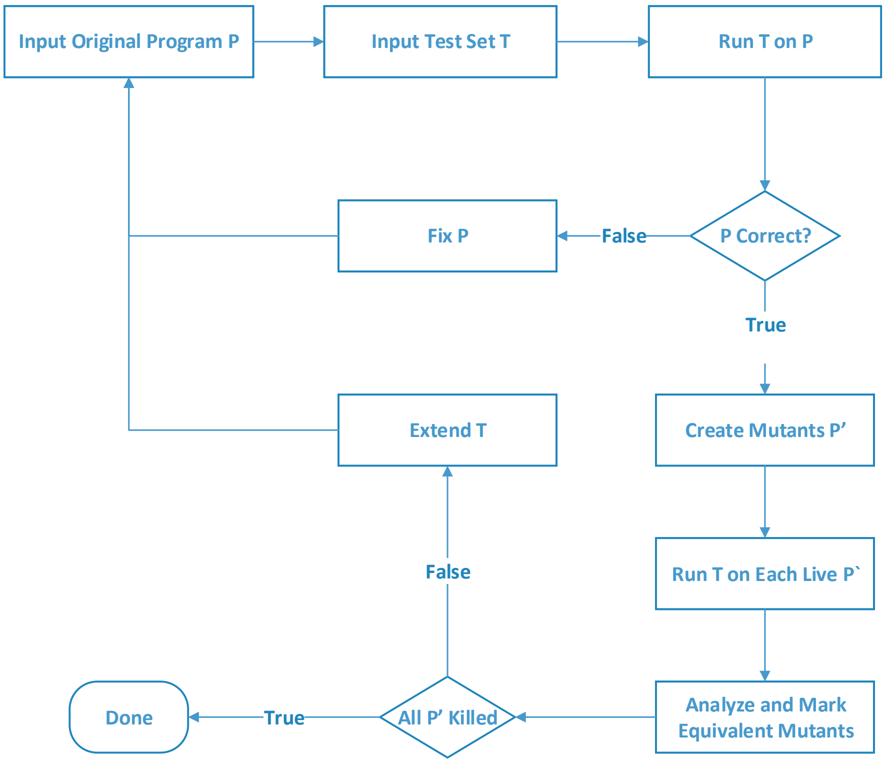

Fig. 1. Mutation analysis process adopted from [26]

Jia [23] provides an apt description of mutation testing: "Mutation testing is a fault-based testing technique which provides a testing criterion called the "mutation adequacy score". The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults".

The Process. The standard mutation testing process is described in Fig. 1. Initially, a set of mutants p' is generated based on a set of mutation operators. Mutation operators are transformation rules that apply a syntactic operation to the original program p to generate the mutated program p'. As mentioned in the previous section, an example of such a mutation rule is to substitute "$>$" by "$<=$".

The original program is then checked for correctness. The correctness of p under the test suite is required, as it is not possible to distinguish a program p from a mutated program p' using a test suite that classifies p as faulty.

If the original program is correct, then the test suite is applied to the different mutants. Executing the test suite on one particular mutant can lead to different results: Alive, Killed, Timed Out and Error. The following are rules used to determine the state of a mutation:

- All tests succeeded $\rightarrow$ Alive
- At least one of the test fails and the mutant is discovered $\rightarrow$ Killed
- Execution of the test suite does not terminate $\rightarrow$ Timed Out
- An error is encountered in the execution of the test suite $\rightarrow$ Error

The last two classes, Error and Timed Out, are required because mutation operators can generate mutated code that does not compile or that includes infinite loops. These cases are not considered for the mutation score as they do not show the fault detection efficiency of the test suite.

In calculating the mutation score, it is also important to consider that a mutation operator can generate a mutant that is syntactically different, and semantically equivalent to the original program. These mutants do not introduce a fault in the program, and therefore, should not be considered in the mutation score. Formally, a mutant of this kind is referred to as an "Equivalent" mutation. Classification of mutants as "Equivalent" can require manual inspection of the surviving mutants.

Applying the before mentioned rules to the results of each test run, allows us to calculate the mutation score, which is the ratio between the total number of detected faults and the total number of non-equivalent seeded faults that did not end in the timed out or error state. A high score implies that the test suite is effective at detecting faults, a low one suggests that adaptions should be made to the test suite.

Coupling Effect. The Coupling Effect was proposed by DeMillo et al. [16]. It states that "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors". This effect allows us to limit our mutation operators to only consider simple syntactic changes, as the tests killing simple errors would also kill more complex errors. Extending the analysis to include mutation operators that introduce complex changes will only increase the number of mutants to test and should not affect the mutation score. Thus, one can assume that test cases unable to detect simple mutations, also are more likely to permit more complex errors to remain undiscovered. This insight adds value to mutation testing as a technique for bug finding, rather than just for code and test quality.

2.2 Ethereum Smart Contracts

Ethereum. Similar to the well-known blockchain protocol Bitcoin, Ethereum is a blockchain that can keep track of transactions and balances.

A key aspect of Ethereum is its ability to have contract accounts, in addition to regular user accounts. These contract accounts have code associated with them, and every time a transaction is sent to such an account, the nodes in the Ethereum network evaluate this code. These contracts allow the implementation of co-owned wallets or the governance of an organisation. The interactions with these contracts are, because of the decentralised and open nature of the Ethereum network, less prone to problems like corruption and secrecy.

Smart contracts have a few quirks and properties which make them both useful and challenging to secure. The first property is that all smart contracts are uploaded to a public blockchain, which means that the code, is immediately available to all adversaries, even if it is only bytecode.

The second aspect is that Ethereum smart contracts that are in use can hold tremendous amounts of value, and even small bugs allow attackers to freeze[1] or steal the currency stored in the contract.

This has led to the implementation of several security analysis tools, which automatically check for the presence of known weaknesses and vulnerability patterns. These techniques aim to improve the security of the software by looking for patterns that are known to be vulnerable.

Truffle. Truffle is a framework that aims to provide a development environment for Solidity developers. For the purpose of this paper, its most relevant aspect is that it allows developers to write and execute unit tests. These tests can be written in Javascript using the Mocha framework, or in pure Solidity. Truffle then uses another tool called Ganache, a Javascript Ethereum node which supports additional JSON RPC calls for the purposes of testing, to execute these unit tests.

3 Design

To evaluate the feasibility of mutation testing in the context of Ethereum-based smart contracts, we created Vertigo, a mutation testing framework for smart contracts. Vertigo is available on Github[2]. Vertigo will be made available as open source.

This section describes the design and implementation of Vertigo. Figure 2 shows the structure and interaction of the base components in Vertigo. The three main sections shown in Fig. 2 are discussed in order.

[1] With *freeze*, we mean the act of blocking users from accessing the currency stored in the contract.

[2] https://github.com/JoranHonig/vertigo.

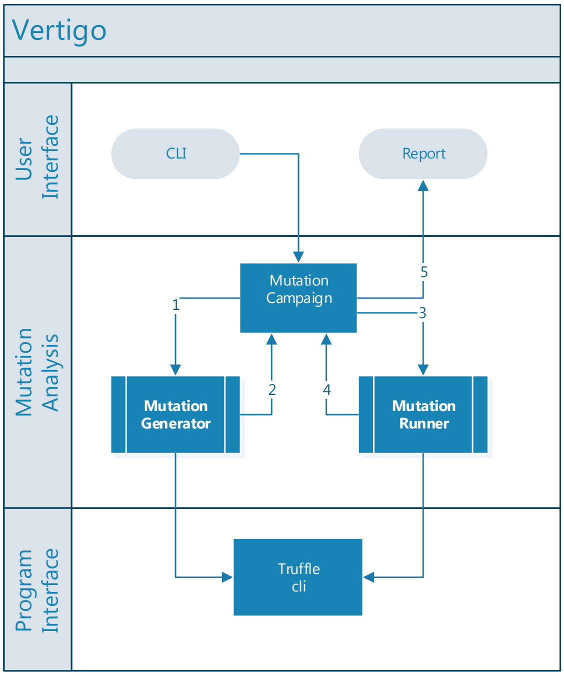

Fig. 2. Structure diagram for components in Vertigo

3.1 User Interface

Through the command line interface (CLI) users are able to initialise mutation testing campaigns. It is possible to configure the Ethereum networks that will be used for the test runs, through the use of command line options. Using a command line option, Vertigo can also be instructed to apply the mutation sampling technique, which will be elaborated in Sect. 3.2.

By default, Vertigo will report all mutants that have survived a mutation campaign. Optionally, a full report with the other mutations, can be written to a file.

3.2 Mutation Analysis

The mutation analysis compartment is where most of the work happens, it is comprised of three interconnected components. These components implement everything from generating mutants to directing the testing process. The edges visible in Fig. 2 are numbered according to the data flow through the mutation analysis components.

Table 1. Increments mirror substitution matrix

Original	Result
+=	=+
−=	=−

Table 2. Mutation operators

Mutation operator	Description
Condition Boundary	Replaces an conditional operation with its inclusive or exclusive counterpart
Condition Negation	Replaces a conditional operation with its inverse
Math Inversion	Replaces a math operator with its inverse
Increments Inversion	Replaces an increments statement with its inverse

(a) Mutation operators inherited from PIT

Mutation operator	Description
Increments Mirror	Replaces an increments statement with it's mirror
Modifier Removal	Removes a modifier application

(b) Vertigo mutation operators

Mutation Campaign. The mutation campaign is the central component in Vertigo that directs and manages the mutation testing of a project. What follows is an overview of the steps taken by the mutation campaign while performing mutation testing.

The very first step is the application of the test suite on the original program. If the given test suite fails on the original contracts, then further analysis is halted, as it will be impossible to determine if the test suite distinguishes a mutated program from the non-mutated program.

During the next step in the mutation testing process, the mutation campaign will request the Mutation Generator component to generate a set of mutations for the project.

The Mutation Campaign implements functionality that allows it to apply custom filters on the set of generated mutations. In Vertigo we implement one initial filter that selects a random sample from the available mutations for further analysis. This is an implementation of the optimisation technique "Mutant Sampling" [23], which is based on the assumption that the mutation score of a random sample of the mutants will reflect that of the entire set of mutations. Wong [29] showed that a sample of 10% of the entire mutation set is only 16% less effective than an analysis of all the mutants, with respect to the mutation score.

To facilitate the implementation of mutant sampling, we have implemented an extensible interface for filters. This same interface can also be used to implement other filtering techniques, such as mutant clustering [22] or equivalent mutation detection [23].

The generated and filtered mutations are passed to the Mutation Runner, which applies the test suite on the mutated contracts storing the result of each test run.

Finally, the mutation campaign will return the results to the user interface, where these results are formatted and displayed to the user.

Mutation Generator. The Mutation Generator component is responsible for the generation of mutants, which are generated through the application of so-called mutation operators. These mutation operators [23] are transformation rules that can be used to generate mutant programs p' from a program p. An example mutation operation could specify the replacement of "==" with "!=". The mutation operators implemented in Vertigo have been both specifically designed for the context of smart contracts (Table 2b) and based on existing work (Table 2a).

PIT [9], a state of the art mutation testing tool that works on Java, has a well-documented set of mutation operators. While it implements a wide range of mutation operators, not all of them are enabled by default. They try to limit the developer's exposure to equivalent mutations, by only enabling the mutation operators which generate lower amounts of equivalent mutations. Vertigo leverages a few selected mutation operators from the PIT default mutation operators; these are visible in Table 2a.

Additionally, we propose and implement the addition of a set of new mutation operators, relevant to the Solidity programming language and smart contracts. They are designed to simulate bugs and security issues applicable to the Solidity programming language. These mutation operators can be found in Table 2b and will be discussed in the following paragraphs.

Increments Mirror. A recent development in Smart Contract security has been the standardisation and formulation of a weakness registry for smart contracts and Solidity [11]. One issue illustrated in this registry [13] describes a typographical error introduced when "=+" is written instead of "+=". In this case the statement is interpreted as an assignment with a unary operator following it, whereas the developer intends to write an incrementation operation. This Mutation Operation tries to introduce the same faults in the target program, substituting incrementations according to Table 1

Modifier Removal. Solidity allows the use of modifiers. These are functions that can be used to wrap the functionality of other functions, similar to decorators in python. One pattern used in smart contract development is that of ownership. A contract that implements this functionality has a variable that stores the address of the account that is currently the owner of the contract, and functionality to transfer ownership. This contract will also implement a modifier which is called onlyOwner; this modifier will change every function that invokes it to revert if the caller is not the owner of the contract. The ownership paradigm, allows smart contract developers to protect administrative functions in a smart contract.

Forgetting to add a modifier is a simple mistake that a developer could easily make. Because of this, and the serious effect it can have on the security of a contract we include a mutation operator which simulates the omission of modifiers.

Mutation Runner. The Mutation Runner component implements the logic that allows Vertigo to apply mutations to the test project and then apply these tests on the project. There are two main aspects of this component that are of interest: Its ability to perform multiple test executions in parallel, and the mutation application and autonomous configuration.

The mutation application process is described by the following steps:

1. Vertigo creates a temporary directory and copies the files from the initial project to it.
2. The mutation is applied to the source code
3. The Truffle configuration is adapted allowing Vertigo to interpret test results
4. The Truffle test command is invoked directed at the mutated project

Because Vertigo uses temporary directories, the original project directory will never need to be written to. The use of temporary directories allows parallel application of multiple mutations; additionally, there is no risk of a mutation persisting if Vertigo fails unexpectedly.

Parallelisation. The user is able to indicate the networks that will be used for the mutation testing process. These networks provide a virtual environment used for the execution of the tests. Parallel testing is enabled if Vertigo is provided with multiple networks, in which case each instance of a runner is dynamically assigned a free network for each test run.

3.3 Program Interface

Vertigo already implements an interface with Truffle and its CLI right now, but this is extensible and allows integration of different platforms with a similar interface.

The first element exposed by the Truffle interface is the analysis of source code and the generation of an abstract syntax tree (AST). The information in the AST is used by the Mutation Generator to find possible mutations.

The second element is the application of the test suite on a project, which allows the Mutation Runner component to check if the test suite distinguishes the original program from a mutated one.

Ganache. Truffle uses Ganache to model an Ethereum blockchain. Ganache extends the traditional Ethereum interface by adding some specific JSON RPC calls that allow Truffle to execute each test in a clean environment. During the development of the Vertigo framework, we found that Truffle, through its support for the Mocha framework, also supports the writing of tests that do not use this clean environment. This results in a situation where although a program

Table 3. Mutation test results

	Lived	Error	Timeout	Killed	Equivalent	Total	Mutation score	Code coverage	Duration
Aragon OS	0	27	1	306	0	334	100%	99%	129 min
Openzeppelin-solidity	30	55	3	303	5	391	92%	100%	260 min

might be correct under a test set T, it might not be under consecutive runs of the test set T. As a solution, we propose that developers refrain from using this functionality. Alternatively, if the only side effect caused by the tests is an alternative distribution of resources over the test accounts initialised by the Truffle framework, then the user can decide to initialise Ganache with the option -e to increase the default balance of the test accounts. This change will increase the number of times that T can successfully be applied.

4 Evaluation

In this section we apply mutation analysis on two popular smart contract projects. Moreover, we provide a discussion on the analysis results provided by Vertigo.

4.1 Experiments

We analysed two popular smart contract projects, both of which boast impressive code coverage results (see Table 3). Since both projects are very well maintained and extensively tested, any analysis results can provide insight into the applicability of mutation testing to real-world smart contract projects.

The first project we analyse is AragonOS, which is described as "a smart contract framework for building decentralised organisations, dapps, and protocols" [1]. The second is openzeppelin-solidity [7], a library of contracts for secure contract development.

The results of these test runs can be found in Table 3. The mutation score is calculated using the following formula:

$$score = \frac{(killed)}{(lived - equivalent) + killed} \times 100\% \tag{1}$$

The experiments were executed on a Ubuntu 18.04 machine with the following available resources: Threadripper 1950X, 32 GB RAM and a 1 TB NVMe SSD. The mutation testing was performed using 16 parallel testing networks.

4.2 Discussion

The results show that AragonOS outperforms openzeppelin-solidity on mutation score even though openzeppelin-solidity would seem to perform better when code coverage is used as a metric. If the developers of either project decide that they want to improve the quality of their test suite, then code coverage will not be able to provide much insight, as both projects have a coverage of 99 or 100%.

However, Table 3 shows that openzeppelin does not have a perfect mutation score; thus, the application of mutation testing would allow the developers to improve their test suite. Besides the mutation score, the developers can also use the surviving mutations to identify the portion of code that requires improved tests.

Additionally, the 30 surviving mutations for openzeppelin-solidity demonstrated yet uncovered edge cases in the behaviour of the program. An example of such an edge case is a function which uses a Solidity modifier to implement access control. The survival of a mutant that removes such a modifier, indicates that the test suite might not check whether an access control policy is enforced for this function. Without an automated technique like mutation testing, discovering these edge cases would have been strenuous and time-consuming.

While the results show that mutation testing can be used to improve openzeppelin-solidity, the analysis was unable to find a mutant which would not be caught by the test suite of Aragon OS. This result indicates that the developers of this project were able to implement a high-quality test suite without the help of mutation testing.

During the inspection of the analysis subjects, we realised that the smart contract projects sparingly made use of plain arithmetic operations. Rather, the projects employ the use of safe math libraries. A safe math library is a library that implements simple arithmetic operations, like addition and subtraction. These libraries extend normal arithmetic behaviour with safety checks. For example, the add functions will check if the result of an addition is overflowed and raise an exception that is the case. As a result, most of the business logic will use functions like add() and subtract(), instead of the binary operators $+$ and $-$. This can have an impact on some of the mutation operators which target mathematical operators, as the usage of plain arithmetic operators will be less frequent. Introduction of mutation operators that extend the behaviour of the arithmetic mutation operators to the safe math functions is a topic for future work (see Sect. 6.1).

Finally, the results in Table 3 show that a remarkably low number of equivalent mutants have been generated. This shows the effectiveness of the mutation operators taken from PIT [9] (visible in Table 2a). Additionally, there were no equivalent mutants for the mutation operators in Table 2b. The low amount of equivalent mutants being generated by the mutation operators in Vertigo is beneficial for the application of mutation testing, as the manual process of determining equivalency for smart contracts can be time-consuming.

5 Related Work

Securing smart contracts is a difficult task, for which multiple approaches have been proposed and implemented. These tools apply techniques such as symbolic execution (Mythril [6], Oyente [24], Maian [25], Manticore [5]), or perform data and control flow analysis (Securify [27], Slither [10], Vandal [15]). A common aspect of these tools is that they leverage common patterns that describe vulnerabilities, to report on the existence of vulnerabilities or bugs. This makes them complementary to the application of unit testing and mutation testing, as they target the identification of common bugs, whereas unit testing allows for the testing of business logic.

Besides fully automated approaches, there are also verification approaches where a user is asked to define invariants and properties which are then validated (K-framework [21] and Verisol [28]). Such approaches are complementary to mutation testing, as mutation testing can also be applied to the evaluation of specifications [17].

There have also been some developments in the application of mutation testing to smart contracts. Eth-mutants [4] is a project that implements a proof of concept mutation analyzer for Solidity. However, key optimisations like parallelisation are not present in this project, such optimisations are necessary to make mutation testing practical for real-world projects. Additionally, Vertigo is designed to be extensible with various proposed optimisations [23] in mind.

Another project is Universalmutator [20], which has recently been extended with mutation rules for Solidity. Universalmutator is a project aimed at rapid development of mutations for different programming languages, using regular expression based substitution rules. This approach is focused on the generation of mutants, whereas Vertigo aims to implement the entire mutation testing process.

6 Conclusion

In this paper, we studied the design of a mutation testing framework and the application of mutation testing to the domain of smart contracts. We showed that mutation testing allows developers to both gauge the effectiveness of their test suite and improve the test suite in order to increase this effectiveness. Additionally we provide analysis results for two major smart contract projects, openzeppelin-solidity [7] and AragonOS [1]. For this paper, we created the tool Vertigo, an implementation of the proposed designs.

6.1 Future Work

The implementation of Vertigo relies upon functionality exposed by Truffle through its command line interface. This interface does not yet support running individual tests and code coverage reporting. These features can provide essential information for optimisations that allow Vertigo to "do less" [23].

Specifically, the addition of these features will allow for the implementation of the following two optimisations, which are also available in PIT [9]:

– Instead of running the entire test suite for each mutant, Vertigo should determine the specific tests that cover the line on which the mutant introduces a syntactic change. The results of tests that do not cover this line should not depend on the change, and therefore these tests provide little value when executed on the mutant.

– Instead of running the tests in the test suite in a random or pre-set order, Vertigo should optimise the order, so fast tests are executed earlier. Once a test fails, the execution of the other test can be omitted since they do not provide additional value over the previously executed test.

For this paper, we selected a limited set of mutation operators to provide a prototype that is applicable to real-world projects. A topic for further research is to evaluate existing mutation operators, and design new mutation operators specifically for the analysis of smart contracts. An evaluation of mutants with respect to their representativeness of the mutation score allows for the application of mutant selection as an optimisation technique.

Vertigo now supports mutation of Solidity smart contracts, and leverages the Truffle framework to execute the tests; thus, Vertigo is limited to the mutation testing of Solidity smart contracts for ethereum. A possible extension to Vertigo is the design of mutation operators for other smart contract languages and the implementation of interfacing logic for other test frameworks. Such an extension allows for a more uniform application of mutation testing over different blockchain platforms.

Furthermore, examining security critical bug classes like those described in the SWC registry [11] to design mutation operators that try to introduce vulnerabilities is promising. Daran and Thévenod-Fosse [18] have examined to what extent seeded faults represent actual faults. A possible topic for further research is to measure the extent to which mutation operators, like *Modifier Removal* in Table 2b, represent actual security critical faults.

Finally, in Sect. 4 we discussed the impact of so-called safe math libraries on the mutation analysis results. Specifically, we saw that the use of plain arithmetic operators was replaced by the use of functions that extend the basic behaviour of these arithmetic operators in the tested projects. Vertigo can be extended to more extensively cover these projects by implementing one of the following approaches:

– the implementation and design of mutation operators that reflect the behaviour of the mathematical operations for commonly used safe math functions
– An extension of Vertigo to allow for project specific mutation operators, allowing the developer to express the mathematical mutation operators for their specific project

Additionally, a new mutation operator can be designed for projects using safe math libraries, that introduces faults that mimic developers forgetting to use safe math libraries; a problem that has resulted in the well known “batchOverflow” bug [2].

References

1. aragonOS. https://hack.aragon.org/docs/aragonos-intro.html
2. Batch overlflow vulnerability - CVE-2018-10299. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10299
3. CryptoKitties. https://www.cryptokitties.co/
4. eth-mutants: a mutation testing tool for smart contracts. https://github.com/federicobond/eth-mutants
5. Manticore. https://github.com/trailofbits/manticore
6. Mythril. https://github.com/consensys/mythril
7. openzeppelin-solidity. https://github.com/OpenZeppelin/openzeppelin-solidity
8. Parity Bug Security Alert. https://www.parity.io/security-alert-2/
9. PIT Mutation Testing. http://pitest.org/
10. Slither: Static Analyzer for Solidity. https://github.com/crytic/slither
11. Smart Contract Weakness Classification and Test Cases. https://smartcontractsecurity.github.io/SWC-registry/
12. Solidity. https://github.com/ethereum/solidity
13. SWC-129. https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-129
14. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft - CoinDesk. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft
15. Brent, L., et al.: Vandal: a scalable security analysis framework for smart contracts. CoRR (2018)
16. Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F.G.: The design of a prototype mutation system for program testing. In: Proceedings of the AFIPS National Computer Conference, vol. 74, pp. 623–627 (1978)
17. Budd, T.A., Gopal, A.S.: Program testing by specification mutation. Comput. Lang. **10**(1), 63–73 (1985). https://doi.org/10.1016/0096-0551(85)90011-6
18. Daran, M., Thévenod-Fosse, P.: Software error analysis. In: Proceedings of the 1996 International Symposium on Software Testing and Analysis - ISSTA 1996, vol. 21, pp. 158–171. ACM Press (1996). https://doi.org/10.1145/229000.226313
19. Dijkstra, E.W.: Ewd 249 Notes on Structured Programming, 2nd edn. Department of Mathematics, Technische Hogeschool Eindhoven (1970)
20. Groce, A., Holmes, J., Marinov, D., Shi, A., Zhang, L.: An extensible, regular-expression-based tool for multi-language mutant generation. In: Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE 2018, pp. 25–28. ACM Press (2018). https://doi.org/10.1145/3183440.3183485
21. Hildenbrandt, E., et al.: KEVM: a complete semantics of the Ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium, pp. 204–217. IEEE (2018). https://doi.org/10.1109/CSF.2018.00022
22. Hussain, S.: Mutation clustering. Master's thesis, King's College London, UK (2008)
23. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
24. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS 2016, pp. 254–269. ACM Press, New York (2016). https://doi.org/10.1145/2976749.2978309

25. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference. ACSAC 2018, pp. 653–663 (2018). https://doi.org/10.1145/3274694.3274743
26. Offutt, A.J., Untch, R.H.: Mutation 2000: uniting the orthogonal. In: Wong, W.E. (ed.) Mutation Testing for the New Century, pp. 34–44. Springer, Boston (2001). https://doi.org/10.1007/978-1-4757-5939-6_7
27. Tsankov, P., Dan, A., Cohen, D.D., Gervais, A., Buenzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018 (2018). https://doi.org/10.1145/3243734.3243780
28. Wang, Y., et al.: Formal specification and verification of smart contracts for Azure blockchain. CoRR (2018)
29. Wong, W.E.: On mutation and data flow. Ph.D. thesis (1993)