DEVIANT: A MUTATION TESTING TOOL FOR SOLIDITY SMART CONTRACTS

by

Patrick Chapman

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2019

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Patrick Chapman

Thesis Title: Deviant: A Mutation Testing Tool for Solidity Smart Contracts

Date of Final Oral Examination: 1 July 2019

The following individuals read and discussed the thesis submitted by student Patrick Chapman, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Dianxiang Xu, Ph.D.	Chair, Supervisory Committee
Gaby Dagher, Ph.D.	Member, Supervisory Committee
Jyh-haw Yeh, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Dianxiang Xu, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Dr. Dianxiang Xu, for guiding me during this research. Without their guidance, I would have struggled greatly with understanding the concepts that went into my research. I would also like to thank Dr. Lin Deng and his student Yin Xiong from Towson University. Dr. Deng had provided his expertise in mutation testing, while him and Yin have provided useful testing resources for these Solidity projects, making this research possible.

I would also like to thank my committee members Dr. Jyh-haw Yeh and Dr. Gaby Dagher. Their advice on how to strengthen my research was incredibly useful.

ABSTRACT

Blockchain in recent years has exploded in popularity with Ethereum being one of the leading blockchain platforms. Solidity is a widely used scripting language for creating smart contracts in Ethereum applications. Quality assurance in Solidity contracts is of critical importance because bugs or vulnerabilities can lead to a considerable loss of financial assets. However, it is unclear what level of quality assurance is provided in many of these applications.

Mutation testing is the process of intentionally injecting faults into a target program and then running the provided test suite against the various injected faults. Mutation testing is used to evaluate the effectiveness of a test suite, measuring the test suite's capability of covering certain types of faults. This thesis presents Deviant, the first implementation of a mutation testing tool for Solidity smart contracts. Deviant implements mutation operators that cover the unique features of Solidity according to our constructed fault model, in addition to traditional mutation operators that exist for other programming languages.

Deviant has been applied to five open-source Solidity projects: MetaCoin [22], MultiSigWallet [23], Alice [21], aragonOS [24], and OpenZeppelin [25]. Experimental results show that the provided test suites result in low mutation scores. These results indicate that the provided tests cannot ensure high-level assurance of code quality. Such evaluation results offer important guidelines for Solidity developers to implement more effective tests in order to deliver trustworthy code and reduce the risk of financial loss.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

ABSTRACT v

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF ABBREVIATIONS xi

CHAPTER ONE: INTRODUCTION 1

CHAPTER TWO: BACKGROUND AND RELATED WORK 4

- 2.1 Blockchain 4
- 2.2 Ethereum and Solidity 6
- 2.3 Quality Assurance for Smart Contracts 10
- 2.4 Mutation Testing 19
  - 2.4.1 muJava 22
  - 2.4.2 Stryker 23

CHAPTER THREE: THE DESIGN OF DEVIANT 25

- 3.1 Overview 25
- 3.2 Design Challenges 27
  - 3.2.1 Program Validity 27
  - 3.2.2 Language Evolution 28
  - 3.2.3 Ethereum Evolution 29

3.2.4 Reasonable Operators 30

3.2.5 Performance 30

3.3 Fault Model of Solidity Smart Contracts 32

3.4 Mutation Operators 35

3.4.1 Intra-Statement Level 35

3.4.2 Intra-Function Level 39

3.4.3 Intra-Module Level 42

3.4.4 Inter-Module Level 46

3.4.5 Mutation Operator Relationship to Fault Types 47

CHAPTER FOUR: IMPLEMENTATION 51

4.1 Implementation of Deviant 51

4.2 Implementation Issues 57

CHAPTER FIVE: EMPIRICAL EVALUATION 60

5.1 Experiment 60

5.2 Analysis 66

5.2.1 Gas Mutants 67

5.2.2 Address Mutants 67

5.2.3 Address Function Mutants 68

5.2.4 Data Location Mutants 69

5.2.5 Event Mutants 69

5.2.6 Selfdestruct Mutants 70

5.2.7 Exception-Handling Mutants 71

5.2.8 Modifier Mutants 72

5.2.9 Function Type Mutants 73

5.2.10 Function Visibility Mutants 73

5.2.11 Library Function Visibility Mutants 74

5.2.12 State Variable Visibility Mutants..74

5.3 Threats to Validity 75

CHAPTER SIX: CONCLUSION 77

REFERENCES 78

APPENDIX A 82

A.1 General Intra-Statement Level Mutation Operators.......................................83

LIST OF TABLES

Table 3.1 Address Mutation Operators 36

Table 3.2 Address Function Mutation Operators 38

Table 3.3 Selfdestruct Mutation Operators 40

Table 3.4 Exception-Handling Mutation Operators 41

Table 3.5 State Variable Mutation Operators 42

Table 3.6 Function Type Mutation Operators 43

Table 3.7 Function Visibility Mutation Operators 44

Table 3.8 Library Function Visibility Operators 46

Table 3.9 Fault Types with Associated Mutation Operators 47

Table 5.1 Subject Program Metrics 62

Table 5.2 Subject Programs and Their Contracts Used for Mutation 63

Table 5.3 Subject Program Experimental Results Including Potential Equivalent Mutants 65

Table 5.4 Number of Live Mutants for each Mutant Type of Solidity-Specific Features 66

Table A.1 Binary Mutation Operators 84

Table A.2 Unary Mutation Operators 85

Table A.3 Assignment Mutation Operators 86

Table A.4 Literal Mutation Operators 87

Table A.5 Intra-Module Level Mutation Operators 90

LIST OF FIGURES

Figure 2.1 Blocks in the Bitcoin blockchain model [12] 5

Figure 2.2 State transition model in the Ethereum blockchain model [12].................. 8

Figure 3.1 Architecture of Deviant.. 26

Figure 4.1 Home Page for Deviant GUI.. 52

Figure 4.2 Mutation Operators Page for Deviant GUI.. 53

Figure 4.3 Report Page for Deviant GUI 54

LIST OF ABBREVIATIONS

EVM	Ethereum Virtual Machine
dApp	Decentralized Application
AST	Abstract Syntax Tree
UTXO	Unspent Transaction Output
GUI	Graphical User Interface
JSON	JavaScript Object Notation
FSM	Finite State Machine

CHAPTER ONE: INTRODUCTION

Blockchain is an increasingly popular technology that led to one of the biggest economic anomalies in recent years. As a popular platform for blockchain applications, Ethereum provides a Turing-complete instruction set [12], allowing for the development of more computationally expressive smart contracts in blockchain applications. Among the several high-level programming languages that can compile into Ethereum Virtual Machine (EVM) bytecode, this thesis focuses on Solidity. Different from traditional programs, a Solidity smart contract cannot be simply patched after it has been compiled and added to the blockchain because of the nature of Ethereum and blockchain. Quality assurance of smart contracts is therefore extremely important in the development process. For example, a vulnerability in the DAO (decentralized autonomous organization) resulted in the loss of approximately $50 million worth of ether (Ethereum's cryptocurrency) [36].

As for traditional software development, software testing is one of the common techniques for quality assurance in any programming language, Solidity included. Software testing exercises a Solidity program with test cases, aiming to find faults or vulnerabilities. Most of the existing popular open source Solidity projects have included built-in tests. Nevertheless, it is unclear what level of code quality can be assured by these tests or how effective the tests can be. One way to assess the testing effectiveness is to evaluate how many types of faults that can be revealed. In reality, however, real-world projects seldom keep track of every fault that has occurred during their development processes. To address this issue, a widely-applied approach is mutation testing, which aims to simulate programming faults by creating mutants of a given program [40]. Each mutant has one fault injected by a mutation operator. A mutant is said to be killed if it fails one or more test. A live mutant not killed by any test can be either faulty or equivalent to the original program. Mutation score, i.e., the mutant-killing ratio between the number of mutants killed and the total number of non-equivalent mutants, is often used to indicate the fault detection capability of given tests. Mutation testing has been applied to various programming languages, such as Java, C, and JavaScript. Experiments have shown that mutants are indeed similar to real faults for the purpose of evaluating testing techniques [41].

In this paper, we present Deviant, a mutation testing tool for Solidity smart contracts. It aims to automatically generate mutants of a given Solidity project that simulate various faults that may occur during the programming process, and automatically run the test cases of the given Solidity project against each mutant so as to evaluate the effectiveness of the given tests. Thus, Deviant can help Solidity developers deliver higher quality code, reduce the risk of financial loss, and increase user satisfaction.

This research is the first attempt to apply mutation testing to Solidity smart contracts. The contributions are twofold:

- In addition to the mutation of traditional programming constructs (e.g., expression and inheritance), Deviant covers all the features that are unique to Solidity smart contracts. This allows for the evaluation of tests that target Solidity-specific features.

- Deviant has been applied to five Solidity programs to evaluate the effectiveness of their tests. The results indicate that these tests have not achieved high mutation scores and thus cannot provide a high-level assurance of code quality.

The rest of this paper is organized as follows. Chapter Two introduces background information and related work on blockchain, Ethereum/Solidity, quality assurance for smart contracts, and mutation testing. Chapter Three describes the design of Deviant, focusing on the fault model and Solidity-specific mutation operators. Chapter Four discusses the implementation. Chapter Five presents the experimental results and provides analysis. Chapter Five concludes the paper.

CHAPTER TWO: BACKGROUND AND RELATED WORK

Chapter Two presents the background information and related work for blockchain, Ethereum/Solidity, quality assurance of smart contracts, and mutation testing. Section 2.1 gives the background of blockchain and explains why it is a growing trend among software developers. Section 2.2 explains Ethereum and Solidity, highlighting uniqueness and differences that exist between Ethereum and the earliest form of blockchain. Section 2.3 reviews related work in quality assurance for smart contracts and how it relates to our work. Finally, subsection 2.4 concludes with an overview of mutation testing and its purpose.

2.1 Blockchain

Blockchain is a data structure comprised of records that are linked together using cryptography [1]. The blockchain can be described as a decentralized distributed database or ledger. A blockchain is considered decentralized due to the fact that there is no central authority that controls what happens on the blockchain. The blockchain is controlled by the nodes that participate in the blockchain network, where each node that is participating has a copy of the blockchain. This means that if a malicious user were to try and fake parts of the blockchain, it would be easily considered invalid as everyone else on the blockchain has access to all the records that exist. Because every node on the blockchain has access to the records that exist, all of the nodes in the network are connected to one another, which is why the blockchain is considered distributed. A blockchain can be thought of as a database because the blockchain allows for the storage and retrieval of data, while it is also considered a ledger because information is always being appended to the blockchain. This means that information is constantly being added to the blockchain, never removed or altered. This feature of the blockchain is called immutability and is crucial for confirming the history of transactions that actually take place on the blockchain. The earliest form of blockchain, Bitcoin, was invented by an anonymous user going by the alias Satoshi Nakamoto [2].

Figure 2.1 shows blocks in the Bitcoin blockchain model, where the block contains the information such as: the time the block was created, the unique nonce number, the hash of the previous block, and the list of the transactions that have taken place since the previous block was added.

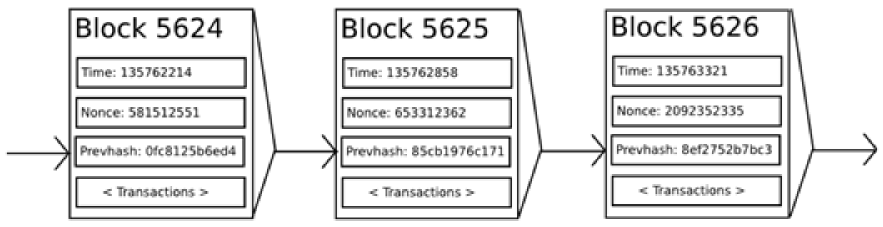


Figure 2.1 Blocks in the Bitcoin blockchain model [12]

Blockchain technologies use scripting languages for their transactions, including Bitcoin. While Bitcoin might be extremely relevant to this day, it still has several major weaknesses with its own language, Script. The first noticeable difference between Script and other common programming languages is that it is not Turing-complete, meaning that it does not support loops or recursion. This feature of Bitcoin was an intentional design choice, as Bitcoin wanted to prevent the halting problem that exists in Turing-complete languages. The halting problem is essentially given a Turing-complete language, how can we determine if the program will actually stop execution. If the

Bitcoin blockchain supported loops or recursion, then it is highly-likely an attack would take place in the form of a Denial of Service (DoS) attack or a programmer error would lead to infinite execution. With the removal of loops and recursion, Bitcoin can ensure that the script execution will eventually stop.

Blockchain also uses a specific record keeping model called Unspent Transaction Output (UTXO). The UTXO model is named as such because output from previous transactions are used as input for new transactions. What this means is that when a transaction takes place, the entirety value of the user's UTXO is used as input for the transaction. The designated amount of Bitcoin is sent to the given address, where the remaining value from the Bitcoin UTXO is actually sent back to the sender's address in the form of a new UTXO.

Bitcoin is however not the only blockchain platform that exists. There have been many blockchain platforms that have been introduced since Bitcoin's inception, most of which have made an effort to improve on some of the concepts behind Bitcoin. In some cases, these new blockchain platforms introduce new ideas entirely or are made specifically for certain purposes.

2.2 Ethereum and Solidity

Ethereum is one of the most popular blockchain platforms that exist today, mostly due to its own innovative design decisions that have differentiated it from Bitcoin. There are several languages that can compile to Ethereum Virtual Machine (EVM) bytecode; however, the most popular of all the scripting languages for Ethereum is Solidity. This section introduces the Ethereum blockchain model and also introduces the scripting language Solidity.

The philosophy behind Ethereum is to provide a blockchain with an instruction set that is simple, universal, modular, agile, and non-discriminatory [12]. Ethereum needs to be simple so that an average programmer can implement a specification without trouble. In a similar vein, Ethereum must be universal such that if the program can be mathematically defined, it should be able to be implemented within Solidity. Ethereum must be also modular so that the separate components of Ethereum are self-contained, while also being agile so that if any of these modules need to be updated, it can be. Finally, Ethereum needs to be non-discriminatory such that Ethereum should not actively try to restrict specific categories of usage. There are the primary concepts that were considered in the design of the Ethereum blockchain.

The Ethereum model, unlike Bitcoin, is based on the account model. Ethereum accounts contains key data values that make the Ethereum protocol work. Like Bitcoin, Ethereum accounts contain a nonce, which is a counter to make sure that transactions can only happen once, meaning that the nonce should be unique. Ethereum accounts also contain the balance for that specific account, where the currency that Ethereum uses is known as Ether. This is different from the Bitcoin model, which does not necessarily have a UTXO matching directly to an account. Another important aspect of Ethereum is that an account may optionally contain contract code. This means that there are two types of accounts for Ethereum, externally owned and contract codes. The externally owned accounts are controlled by private keys, whereas the contract codes are controlled by the contract code that is stored on the account.

Transactions are an important part of Ethereum that refer to the signed data package that stores a message to be sent from an externally owned account. The transactions contain: the recipient of the message, a signature of the sender, the amount of ether to transfer from sender to recipient, an optional data field, a STARTGAS value that is representative of the maximum number of computational steps the transaction execution is allowed to take, and a GASPRICE value that represents the fee the sender pays per computational step. The STARTGAS and GASPRICE values are incredibly crucial to prevent DoS attacks. As mentioned previously with Bitcoin, a blockchain is susceptible to DoS attacks if code running on the blockchain causes infinite execution. Bitcoin prevented this by not implementing a Turing-complete language. However, since Ethereum is considered Turing-complete, Ethereum uses gas to prevent this problem. Gas is a resource that is used by contracts that are executing code and when the gas runs out, then the code will stop executing. An example of an Ethereum state transition is given in Figure 2.2.

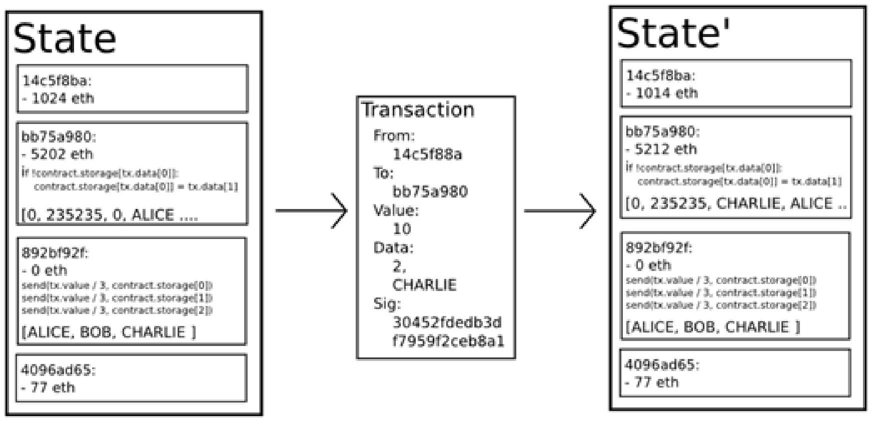

State
14c5f8ba:
- 1024 eth
bb75a980:
- 5202 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, 0, ALICE ....
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth
Transaction
From:
14c5f88a
To:
bb75a980
Value:
10
Data:
2,
CHARLIE
Sig:
30452fdedb3d
f7959f2ceb8a1
State'
14c5f8ba:
- 1014 eth
bb75a980:
- 5212 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, CHARLIE, ALICE ..
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth

Figure 2.2 State transition model in the Ethereum blockchain model [12]

There are currently several scripting languages that exist that can be compiled into EVM bytecode that can be deployed to the Ethereum blockchain. In fact, any developer can create their own language as long as they write a compiler that can compile the language into valid EVM bytecode. The most popular of these scripting languages is Solidity. Solidity is a scripting language that is designed to be simple and rather intuitive for most programmers. To achieve these features, Solidity is designed to closely resemble JavaScript, sharing many of the same basic programming constructs and syntax. Solidity is commonly used to create decentralized applications (dApps), which are applications that take advantage of the inherent features that come with the blockchain model. While it may commonly be assumed that blockchain can only be used for cryptocurrency, it has also been demonstrated to be useful in several areas. In fact, there have been several real-world and several proof-of-concept projects that have demonstrated the usefulness of blockchain in fields such as online voting, healthcare, and governance systems. In many cases, Solidity is often used as the language for implementing these projects because of its simplicity and similarity to traditional programming languages.

Ethereum, while being innovative in the field of blockchain, is still susceptible to faults and vulnerabilities. This can be especially troublesome considering that the Ethereum blockchain is immutable, meaning that once contract code has been deployed to the blockchain, it cannot simply be patched. Naturally, Solidity will inherit these faults and vulnerabilities considering that the Solidity code is compiled and deployed to the Ethereum blockchain. However, Solidity has its own faults and vulnerabilities that are unique to itself. This is because the Solidity compiler may potentially create faulty or vulnerable EVM bytecode, given that like all code, isn’t guaranteed to be safe if it is implemented incorrectly. Of course, there is also a potential for Solidity developers to introduce their own faults in their application. Because Ethereum and Solidity are relatively new technologies, there are some concepts that are not entirely understood by many in the Ethereum community. In fact, throughout Ethereum and Solidity's lifespan, there have been several instances where a fault or vulnerability has had a tremendous effect for users of the Ethereum blockchain. For example, a company known as The DAO (decentralized autonomous organization) contained a vulnerability (known as reentrancy) that led to an attacker stealing approximately $50 million [36] worth of ether (Ethereum's cryptocurrency). When these situations happen, it typically requires a hard fork of the Ethereum blockchain. A hard fork occurs when a non-backwards compatible protocol change occurs in the blockchain. This means that nodes (users) that wish to use the latest version of the Ethereum blockchain must update their Ethereum version to be current or else their future transactions will not be valid on the latest version of the blockchain. Because of the consensus requirements of blockchain, these hard forks require a majority vote in favor of the hard fork. This means that performing a hard fork can be quite a nuisance for the Ethereum community if they constantly have to do so. In a similar manner, Solidity can be quite troublesome if there exists faults or vulnerabilities in contract code. When the Solidity code is stored on the blockchain, it is immutable, meaning that they cannot patch a bug if they find it. Because of this, quality assurance for smart contracts is especially important.

2.3 Quality Assurance for Smart Contracts

With smart contracts having the capability of managing a considerable amount of assets, there has been much research done in the area of quality assurance for Ethereum and Solidity smart contracts. In the case of security, there have been many surveys done on vulnerabilities that exist in the Ethereum platform. Atzei et al. [3] conducted a survey on the security on Ethereum, particularly the vulnerabilities and faults that exist in both

Ethereum and Solidity. Atzei provides a fault taxonomy that describes the vulnerabilities and faults, breaking them down into three levels: Solidity, EVM, and Blockchain. Solidity vulnerabilities are the vulnerabilities that exist because of the improper use of the Solidity language by the programmer or by the language itself. EVM vulnerabilities are the vulnerabilities that exist in the instruction set or limitations that exist in the actual implementation of Ethereum. Finally, blockchain vulnerabilities are those that exist because of the limitations of the blockchain model. Atzei also surveyed several attacks that exist on Ethereum particularly the famous DAO attack and others such as other various vulnerabilities that exist in other applications. This survey concludes that there are some key areas that will be researched in the foreseeable future such as verification of smart contracts and low-level attacks. In our research, we primarily attempt to replicate the faults from the taxonomy that we can directly observe in Solidity. These faults are as follows:

1. Call to the unknown: There are two primary ways that a call to the unknown fault can occur. One reason that a call to the unknown fault can occur is if the smart contract developer sends the wrong data to the receiving contract while using the call function. For example, if a developer tries to call a function that doesn’t exist in a contract, then the fallback function for that contract will execute, causing unknown behavior is the developer doesn’t anticipate this. The second reason is that the developer associates a contract with the wrong address or vice versa. The reasoning for the call to the unknown fault is similar, when the data being

sent to the receiving address is received, the receiving contract will not know how to interpret the data, causing the fallback function to execute.

2. Exception disorder: In Solidity and Ethereum, not all exceptions are handled the same. Like all programs, when contract code is being executed, there is a chance for an exception to occur. In the case of Solidity, typically an exception will be thrown. However, it is quite common for developers to incorrectly handle exceptions. For instance, take the send function call in Solidity, which returns false when it fails whereas the function transfer throws an exception on failure. When a developer calls a contract with the send function call, there is a likely chance that they mishandle the exception-case if they do not realize that send is being used. Because send only returns false, a developer might not be able to determine where the failure is happening.
3. Gasless send: There are instances where a developer might run into an out-of-gas exception when they do not expect it. This unexpected exception typically occurs when a developer is using the send function call. This exception is thrown because the send function call is compiled down to the call function call with an empty signature. Now, the send function call always has a gas stipend of 2300, meaning that it only has a limited number of instructions that can be ran before all the gas has been spent. Now, since the signature field is empty, it will invoke the callee’s fallback function, which may potentially have many gas-expensive instructions.

When executing these instructions, if there are too many, then the out-of-gas exception will occur.

4. Type casts: While Solidity does perform type checking, it cannot directly handle checking things such as addresses directly. What this means if that if an address is cast to a contract, the only thing that Solidity does it check the signatures of the functions of the contract that the address is being cast to. The problem is that Solidity does not actually check to see if that contract resides at that address. For instance, an address may cast to the wrong contract, meaning that when the casted contract is called, it will be highly likely that its fallback function is invoked, creating non-deterministic behavior for the developer.
5. Reentrancy: The most devastating vulnerability to date, the reentrancy vulnerability has led to major financial devastation, such as "The DAO" attack that caused losses of approximately $50 million [36][4] at the time of the attack. The reentrancy attack is based on the fact that it is possible to re-enter a non-recursive function before it is terminated. Take the example:

```
contract vulnerable {
 function badfunct(address addr) { //vulnerable code
 addr.call.value(200)();
 }
}

contract attacker {
 function() { //fallback function
 vulnerable(msg.sender).badfunc(this);
 }
}
```

In the above example, the vulnerable contract's function badfunc, sends the given address ether. Now, the attacker contract has its fallback function invoked, which simply calls the same badfunc that was previously called. This process repeats itself several times before it eventually is stopped, but at this point too much of the ether has been taken away. The reason that this happens is that the fallback function is able to call the badfunc again before the execution is able to stop. However, the much bigger problem resides in the call function. The call function will fail during this process; however, it will not propagate the error and as a result will not reverse the transactions that have taken place, except for the very last transaction. Because of this vulnerability, most developers now only use the send and transfer function calls, which are protected against this type of attack. However, there are still instances where a developer has no other choice than to use the call function.

6. Ether lost in transfer: When transferring ether from one address to another, it is incredibly important that everyone knows the correct address or is implementing their code in such a way that there is no error in getting the correct address for sending ether. This is because ether that is sent to the wrong address will likely be sent to an orphan address. An orphan address is an address that does not actually exist on the blockchain. In the case of sending ether, this means that all the ether that is sent will be lost forever and cannot be retrieved.

The previous faults can be mostly targeted in Solidity, however, there are faults that are primarily concerned with just Ethereum and blockchain. These faults cannot be directly targeted in Solidity, but rather are faults that exist because of the design of Ethereum and blockchain in general. However, these faults are important to consider when designing a smart contract, as the smart contract will naturally have these fault properties associated with them. These faults are as follows:

7. Keeping Secrets: In contracts, fields can be public or private. However, even though a field may be private, when a value is set for that field, the transaction must be sent to miners who publish it to the public blockchain. This allows any observers to obtain information that might lead them to being able to figure out the value. Also, while the actual variable information is kept private during execution, the actual contract code is not private, as it resides on the blockchain. This allows attackers to infer information about what may be happening in a contract and allows a white hat approach when trying to attack an application.
8. Immutable Bugs: Once a contract is put onto the blockchain, it can no longer be altered. This means that once the contract is on the blockchain, if it contains bugs, there is no way to fix it. A common approach to working around this problem is deploy the new version of a contract to a new address and while this may work if someone is constantly being kept up to date on the blockchain, it can leave problems for those that still use the old version of the application. This is actually a big enough problem to where the creators of Ethereum had to create an entire new fork in the

blockchain because of the severity of the reentrancy vulnerability, leading to there being two versions of Ethereum now, Ethereum and Ethereum Classic.

9. Unpredictable State: It is impossible to determine the order that the transactions will run for a contract. That means that if the callee invokes a function from a contract, they cannot determine if the contract will be in the same state that it originally was.
10. Generate Randomness: The execution of EVM bytecode is deterministic. As with any deterministic approach to solving a problem, some information can be inferred. This is especially relevant considering that the blockchain is public. Any outsider can attempt to infer something based on what is observed in the blockchain.
11. Time Constraints: Miners can choose the timestamp on a block with a certain degree of arbitrariness. This means that the miner can determine a timestamp that is advantageous to them. This is because many applications use time constraints in order to determine which actions are suitable.
12. Stack Size Limit: If an attacker were to generate a nearly full-stack and then call a victim function, then if the function does not handle exceptions properly, can have adverse effects when the stack has ultimately reached its limit.

A more general research study was done on the bug characteristics that exist in blockchain systems. This research, done by Wan et al. [5], performed an empirical study on eight blockchain platforms, including Ethereum. Wan discovered that semantic bugs were by far the most prevalent bugs that were found in blockchain, for instance, Ethereum had approximately 70% of its bugs related to semantic errors. This can be attributed to the fact that programmers were not able to understand the semantics of blockchain. However, the bugs that took the longest to fix were related to security vulnerabilities. Typically, security vulnerabilities are attributed to problems with the blockchain itself. While security vulnerabilities may on average have taken longer, they were also not as abundant as bugs such as semantic or environment related. This means that the sample sizes were largely different, meaning that more data might reveal a better trend for this data. If anything, this study has led the blockchain community to believe that programmers might tend to struggle with understanding the semantics of blockchain, causing them to introduce errors because of their lack of understanding.

Because of the error-prone nature of programmers and the risks associated with the blockchain there have been numerous research efforts towards formalizing the way that smart contracts should be designed. Mavridou et al. [16] proposed a FSM (Finite State Machine) approach to generating Ethereum smart contracts. Their framework, named FsolidM [35][39], is deigned to bridge the gap between the programmer's understanding of the Ethereum semantics versus what the actual semantics are. As stated in the previous research, the majority of the programming bugs that exist in blockchain applications are related to semantic errors. This research is designed to potentially address these faults. They claim that their framework offers clear semantics and an easy to use GUI for developers to build a FSM that represents their program. They also provide several plugins that allow developers to prevent common vulnerabilities that exist in Solidity applications. Another research paper written by Grishchenko et al. [17] presented the first complete small-step semantics of EVM byte-code. In this research, they used the proof assistant F* [40] to formalize their small-step semantics. Their research and formalization led to the characterization of several security characteristics. They define these characteristics as call integrity, atomicity, and independence from miner-controlled parameters. The call integrity property is related to bugs such as the reentrancy or call to the unknown, while atomicity is related to mishandled exceptions. Independence from miner controller parameters can be split into two separate categories: independence of mutable account state and independence of transaction environment. Independence of mutable account state deals with bugs such as transaction order dependency and unpredictable state, whereas independence of transaction environment deals with timestamp dependency, time constraints, and generating randomness. These classifications are heavily based on the fault taxonomy that was discussed previously and provided by Atzei. The key contribution in this research paper was the actual formalization of these semantics, which as can be seen from the previous research, is an important area of research as the semantics of Ethereum are often not well understood.

As discussed previously, Solidity often deals with financial assets, meaning that it is critical that security vulnerabilities are identified and fixed early. Liu et al. [18] proposed a fuzzing approach to automatically detect reentrancy bugs in smart contracts. In their research they created a framework ReGuard, to iteratively generate random and diverse transactions, which they could then analyze and determine if a reentrancy attack was present in a smart contract. ReGuard takes in the source code for a smart contract and the output contains a report of the reentrancy bugs that were found during the analysis. The actual workflow involves the process of transforming the contract, the fuzzing engine, and the core detector. Essentially, this tool will run using these components potentially revealing if there are any reentrancy bugs, which in their preliminary evaluation, they were able to identify seven new bugs in existing projects.

2.4 Mutation Testing

Mutation testing is a software engineering concept that has existed since the 1970's with the concept being initially conceived by Lipton and implemented by DeMillo et al. [11]. Mutation testing has had several advancements over the years and has been applied to many programming languages. Mutation testing is the process of modifying small portions of code and then running the modified code against tests. Typically, this is thought of as purposely injecting faults into the source code. By doing this, tests will either fail or pass while running the mutant version of the code. If all tests pass the mutant version of the code, then the mutant is marked live, else the mutant is marked as killed. These markings are used to calculate the mutation score. High mutation scores indicate high quality tests because these tests are able to cover the types of faults that are generated through mutation. Mutation testing can then be used to develop new tests, which can target these faults. Mutation operators explain what changes will happen to the source code when a certain attribute in the source code is found. These operators are comprised of the relevant syntax and its possible changes that can be generated. For instance, take the following code example. The relative operator in this example is the '>' operator, which if true causes the program to return A. However, a potential mutant is to mutate the '>' operator to a '<', causing the program to return B if the program was running under the same conditions when it returned A in the original code.

```
public foo(int A, int B) {
 if (A > B) { // original
 return A;
 }
 return B;
}
```

```
public foo(int A, int B) {
 if (A < B) { // mutant
 return A;
 }
 return B;
}
```

Typically, mutants are similar to real programming mistakes and should highlight areas where a test suite is weak. However, there is also the potential for a valid mutant to be created, while still being functionally equivalent to the original code. These mutants are called equivalent and do not offer any insight into the adequacy of a test suite. These are an expensive problem in mutation testing because they will still be tested while not offering anything of value. There is also the problem of generating mutants based off of the operators which are syntactically incorrect, leading the program to not compile. This is an overhead because the incorrect mutants will still have to compile and fail the process, however, these mutants should not count when determining the mutation score. To address some of these issues, a research paper by Schuler et al. [27], the JAVALANCHE framework was introduced, which uses the invariant detection engine DAIKON [28] to help in identifying equivalent mutants. JAVALANCHE first learns the invariants from the original program and then checks for any existing violations of those invariants that exist in the mutants. When it comes to classification of mutants, JAVALANCHE treats the classification slightly different. When a mutant fails a test, then it is considered “killed”, which is typical in most mutation testing tools. However, when a mutant is considered “live”, it is either in one of two categories: non-violating mutants or violating mutants. Violating mutants are mutants that have passed all of the tests, however, they violate the program invariants that were learned from DAIKON.

These violating mutants are more likely to be considered non-equivalent mutants, since they violate the invariant. Whereas, non-violating mutants do not violate any variant, meaning that they do not appear to impact the program with respect to the invariants. In their evaluation of the framework, they discuss three hypotheses. The first hypothesis is that mutants that violate invariants are less likely to be equivalent than mutants that do not violate invariants. In their research, they conclude that there is enough statistical significance to support this hypothesis, however, they only worked with a small sample size. The second hypothesis is that mutants that violate invariants are more likely to be detected by test suites. In their evaluation, they concluded that this hypothesis was true. For instance, in JAXEN, they detected that 98% of invariant-violating mutants are detected versus 44% of non-violating mutants. The third hypothesis is that the more invariants that a mutant violates, the more likely it is to be detected by tests. Again, their evaluation supports the hypothesis, with all seven projects that were evaluated showing that mutants with a higher number of violations resulted in better detection.

Another common problem that exists in mutation testing is the high computational cost. Because of the extreme number of mutants that can be created for even the simplest of operators, many iterations of running of tests can take place. Even one small program file can result in tens to hundreds of valid mutants. Mathur [30] originally discussed the idea of constrained mutation as an alternative to the normal way that mutation testing had already been done. The basic idea behind constrained mutation is to reduce the number of generated mutants, which in turn should decrease the time that it takes to test all the mutants. The first evaluation of this approach came from Offutt et al. [29], which used the term selective mutation to describe Mathur's approach. Typical mutation testing was approximated to be roughly quadratic, whereas Marthur's approach was estimated to be approximately linear. The concept is quite similar, where we simple remove mutation operators based on their generated number of mutants. For instance, removing two different operators would be called 2-selective mutation. The result was quite positive in this research, where removing the most plentiful mutation operators did not drastically affect the mutation score, but instead improved the performance.

In the paper by DeMillo et al. [11], mutation testing was first implemented and used to evaluate the testing methodology when considering complex errors. In this research, the evaluation took place on providing simple-error data to the tested program. By doing this, it was observed that simple-error data was able to kill multiple-error mutants, essentially killing off mutants that were considered to be more complex. According to their research, they believed that these simple techniques were effective because of the coupling effect. Since then, mutation testing has been applied to evaluate even more testing methodologies. Another application of mutation testing, called weak mutation testing was described be Howden [26]. In weak mutation testing, a simple component mutation takes place and subsequently a mutant version of that program is created containing just the one mutated component. For example, a simple component may include such things as arithmetic or boolean expressions. The advantage of weak mutation testing over strong mutation testing is its ability to generalize errors, which is due to the fact that weak mutation makes no coupling assumptions.

2.4.1 muJava

muJava [6][8] is a mutation testing tool for Java, specifically designed for the generation of mutants that target features of object-orientation in Java. muJava generates mutants based on selected mutant operators and selected Java class files. Once the mutants are generated, they are displayed to the user in such a way that the mutants can be compared directly to the original code. While a simple feature, it is important to notify the user the differences that exist between a mutant and the original code. Whenever a mutant passes the test suite and is considered live, the user must be able to determine if they need to design a new test based on the mutant or if the mutant is considered equivalent. In muJava, it is reported that around 5-20% of the generated mutants are considered equivalent. As mentioned previously, one of the major difficulties in mutation testing is determining if the generated mutants are functionally equivalent to the original program. Since the possibility of a generated mutant being equivalent is relatively high, it is important to let the user easily be able to determine if the mutant is actually live.

While many of the programming constructs that can be mutated in Java are relatively simple, the primary unique contribution of muJava is that it also contains mutation operators based on class [9] and inter-class [10] programming constructs. For instance, these mutation operators can mutate things such as the inheritance structure of Java classes. This means that things such as overridden methods can be modified or deleted, potentially leading to serious consequences or undefined behavior. While Java and Solidity are quite different, muJava is an important tool in our research, as many of the OOP mutation operators can be implemented in Solidity (given that Solidity allows for some features of OOP).

2.4.2 Stryker

Stryker [7] is a mutation testing tool for JavaScript. We have found this tool to be useful for our research as Solidity closely resembles and uses many of the same programming constructs as JavaScript. Stryker supports many traditional mutation operators such as literal operators, arithmetic operators, assignment expressions, and many other general operations. In addition to basic mutation operators, Stryker also supports mutation for several of JavaScript's popular frameworks. The major importance of Stryker to this research project is the design of how they create the mutants. In Stryker, the relevant code is parsed and turned into an AST. After the AST has been generated, Stryker then traverses the AST, looking for any relevant nodes that pertain to the defined mutation operators. When a relevant node is found, Stryker then performs the mutation, creating a new mutant. Stryker also breaks it mutation operators into several categories. This is done to provide clarity and to help developers differentiate the difference between the types of operators. This design is useful for Deviant and provides great insight as to how Deviant will approach mutation testing.

CHAPTER THREE: THE DESIGN OF DEVIANT

Chapter three discusses the design of Deviant by giving an overview, explanation of our fault model, and a discussion of the Solidity-specific mutation operators that we have created. Section 3.1 provides an overview as to how our application will work, illustrating the flow of the program. Section 3.2 presents the fault model and explains the fault types in relation to the programming constructs in Solidity. Sections 3.3 details the mutation operators, which are the Solidity-specific mutations that are designed to replicate the Solidity faults that were discussed in the fault model.

3.1 Overview

Deviant aims to automatically generate mutants of a given Solidity project and run the given tests against each mutant to evaluate the testing effectiveness. Figure 3.1 illustrates the architecture of Deviant. Given a Solidity project together with test code, Deviant selects one program file (contract, library, and/or interface) at a time, parses it into an abstract syntax tree (AST), and applies (user-selected) mutation operators to respective nodes of the AST to generate mutants. Mutation operators are defined according to a comprehensive fault model of the Solidity language. In addition to the normal fault types in traditional languages (e.g., expression in JavaScript and inheritance in Java), our fault model considers Solidity-specific features as well as the existing Solidity fault taxonomy mentioned in the related works. Each mutation operator generates one or more mutants by making one change to the given AST. Each mutant is saved into its own new file which contains the mutation.

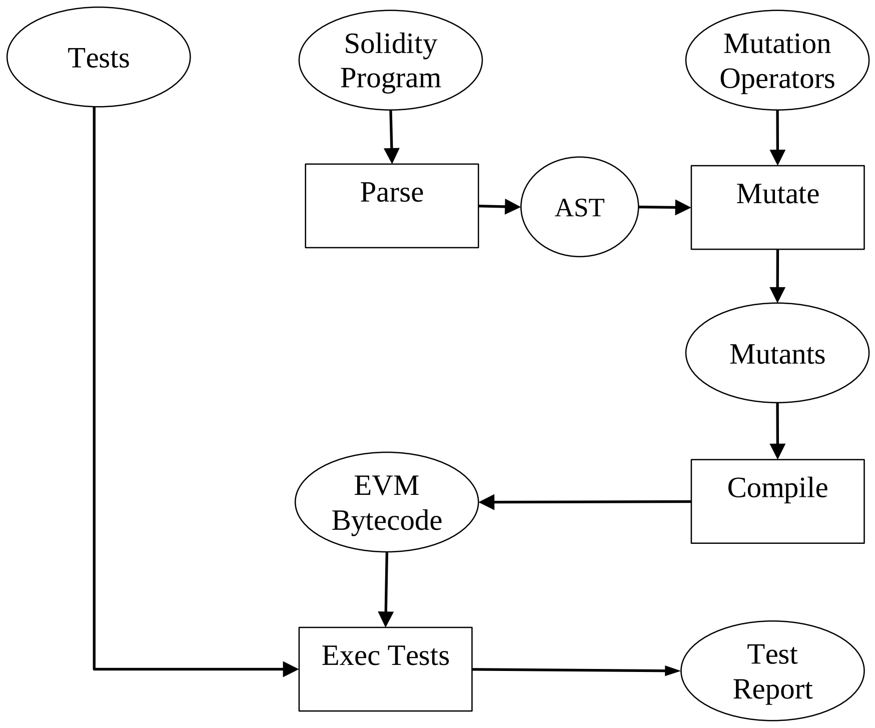


Figure 3.1 Architecture of Deviant

After all the mutants have been generated, a single mutant will be copied into the Solidity project directory and compiled into EVM bytecode as if it were part of the original project. The tests of the given Solidity project will then be run against the mutated EVM bytecode. Tests will either pass or fail depending on how the mutant has affected the functionality of the program. This process is repeated for every single mutant that was generated by Deviant. Deviant keeps track of the test execution result of each mutant (e.g., pass or fail) and produces a summary report on the mutation testing (e.g., mutation score, killed mutant count, and live mutant count). This mutation report gives insight to the Solidity developer on the quality of their test suite. Primarily, the report can highlight weaknesses in the developer’s suite, such as a tendency to miss covering certain Solidity programming constructs.

3.2 Design Challenges

3.2.1 Program Validity

A main technical issue about mutant generation is how to ensure that each mutant is a valid program before compilation. A mutant is only considered valid if it is a contract that can be compiled. In the case of Solidity, we have to ensure both the syntax and semantics will pass the compiler check. The syntax is straightforward as Solidity was designed to be similar to JavaScript. However, the implication of a syntactical change on the semantic is often not as straightforward. There are many instances where a mutation may seem valid at first but ends up failing compilation due to a semantic error. This of course can cause a major performance overhead if many generated mutants are not considered valid. In our design, we can address this issue through precondition checking based on the semantics of Solidity language, particularly the relations between multiple attributes and nodes in the AST. For example, Solidity libraries cannot contain payable functions. Deviant must first check an AST node further up the tree to determine if the function we are working with is contained in a library. A similar constraint on function types is the view or pure function type. When applying a mutation to a view or pure function, we have to consider what the function is modifying or reading in relation to the state of the program. For instance, a pure function can be mutated to a view function, but a view function cannot necessarily be mutated to a pure function since pure functions have the extra limitation of not being able to read the state of the program. However, we are able to determine if a view function can be modified to pure if that function itself does not access state variables and only relies on local variables and the parameters that are passed to the function. It can also be difficult to determine how to deal with abstract contracts in terms of mutation. Abstract contracts in Solidity are never deployed to the blockchain, rather the child contracts that inherit them do. The other problem is that abstract contracts are never directly tagged as abstract with keyword in Solidity. To perform mutation on abstract contracts, we have to determine that at least one function in the contract does not contain an implementation. In Deviant, we can overcome this problem by searching the AST for a function where the body attribute is null, meaning that there is no implementation in the function itself. Once we have done this, we can continue to modify other function bodies through mutation operators.

3.2.2 Language Evolution

Another challenge in designing Deviant is the fact that Solidity is a constantly evolving language. With Solidity being a relatively new language, there are going to be many instances where Solidity features are added, removed, or changed. These changes to the language often times affect the semantics or syntax of the language and can ultimately change the behavior of how previously written code is executed. For example, up until Solidity 0.4.21, constructors could be declared by using the name of the contract as a function name. However, in the more recent versions of Solidity, the keyword constructor must explicitly be used for the creation of a constructor. An even more confusing Solidity change occurred with the introduction of the pure and view keywords that could be associated with function signatures in Solidity 0.4.16. The concept of pure and view were added to Solidity in order to be more semantically meaningful when compared to constant, which is currently just an alias for view. However, the confusing part of this change is that when this version came out, pure functions were not actually functionally “pure”. This meant that while pure functions were described as being functions that were considered “stateless”, they were not actually enforcing this idea completely. Pureness was eventually enforced in Solidity 0.4.17, but still created semantic ambiguity during this time period. There are several other examples of updates in Solidity that have changed what the code actually means.

3.2.3 Ethereum Evolution

A challenge similar to the previous is that the Ethereum blockchain always has potential to change. As developers and researchers continue to use the Ethereum blockchain, it becomes more likely that they will discover new faults with the current design of the Ethereum blockchain. For example, the reentrancy attack was such a crucial vulnerability that a hard fork of the blockchain had to occur. The hard fork in a blockchain platform has serious effects, as it ends up changing the protocol of the blockchain itself. In fact, the nodes that run the old non-forked version of the platform are no longer considered valid when they attempt to make transactions. All of the nodes on the blockchain must upgrade to the latest version of the platform if they wish to continue to operate on the platform. This may seem as if it is a rare occurrence, but it is more common than one may believe. In fact, as of now, there have been seven hard forks since the inception of Ethereum. With the drastic changes that can occur in the protocol, developers have to ensure that their smart contracts that they develop comply with the new semantics that exist in the latest version of the EVM. Beyond the hard forks that have taken place in Ethereum, there have also been many more soft forks that have occurred. The difference between hard and soft forks is that in many cases nodes can have backward compatibility when referencing the newer nodes on the blockchain. However, in many situations these new updates to the blockchain can be quite beneficial to the node user, which means that they must update their own node if they wish to take advantage of these features.

3.2.4 Reasonable Operators

Another more conceptual challenge is designing mutation operators that are considered useful and reasonably targetable by test suites. There are an incredible amount of mutation operators that could be created for any given programming language. However, the challenge is designing mutation operators that are considered useful and reasonable for the developers that use that language. In the case of Solidity, there are several mutants that could easily be generated by Deviant if we chose to do so. However, some of these mutants would be quite meaningless and would practically never be tested in a real test suite as it serves no real purpose. For example, mutating the Solidity version that is defined in the contract will almost never be useful for a developer. In most cases, these contracts won’t even be able to compile if the versions are too different, considering that there may be compatibility issues between the contract versions and the Solidity compiler. This mutation is also likely to never be tested by any smart contract developer, as these types of errors will almost always lead to a compile time error and will actually not ever reach the point of test execution. To overcome this challenge, we only consider the mutation operators that we believe are useful to Solidity developers. The mutation operators that we consider useful are those that can replicate common faults in Solidity and can create behavior in programs that can be caught by test cases.

3.2.5 Performance

A challenge that must be considered in the design of any mutation testing tool is the challenge of performance. Mutation testing is an inherently expensive operation and becomes exponentially more expensive as the project’s complexity increases. For instance, if a test suite takes approximately one minute to run all tests, then that means that each mutant will take one minute to run the entirety of the test suite. Some mutation operators can generate hundreds of mutants for a single program file, meaning that a test suite must be run hundreds of times for all the mutants to be considered. One way that we can alleviate some of this performance problem is to allow the developers to choose the mutation operators that they wish to use during the mutation testing process. While we consider all of the mutation operators to be useful, a developer may only really care about certain types of mutation operators in relation to their test suite. This can be especially useful when removing mutation operators that typically generate the most mutants. For example, one instance of a relational operator can have several mutants involved with the single operator. By ignoring this one mutation operator, we can potentially remove hundreds to thousands of potential mutants across a program. Now, just simply removing mutation operators from the list of applied operators isn’t always the best solution. We can also reduce the number of generated mutants by limiting the number of functionally equivalent mutants that are generated. Functionally equivalent mutants are mutants that are syntactically valid mutants, but are functionally equivalent to the original program, meaning that the mutant will always be considered live because it works the exact same as the original program. As discussed in the mutation testing subsection in the background chapter, equivalent mutants are a complicated problem that has never been completely solved. Currently in Deviant we do not have a great answer to this problem as well. We do not consider equivalent mutants to be part of our total number of mutants when we consider our experimental results, but most of the methodology that has been considered for identifying equivalent mutants cannot be applied to our case yet. The best solution that we have for now is to ensure that we closely follow the semantics that are outlined in the Solidity documentation when creating our mutation operators. By doing this, we can at least ensure that we are at least understanding the semantics of Solidity and can avoid any potential equivalent mutants that may be generated.

3.3 Fault Model of Solidity Smart Contracts

In Deviant, each mutation operator is designed to create mutants that simulate a certain type of faults in Solidity smart contracts. The collection of fault types implied by all mutation operators is referred to as the fault model. The goal of Deviant is to make the fault model as comprehensive as possible so that the generated mutants will simulate as many types of faults as possible. It is worth pointing out that the current mutation operators in Deviant create mutants by making only one change to the original program (called first-order mutation operators). While such small changes may only represent minor faults directly, the mutation testing research has shown that real bugs are often composed of such minor faults [41].

In the following, we describe the fault model from the perspective of Solidity program structures. Generally, a Solidity program consists of version information and three kinds of optional modules (contract, library, and interface), as shown in the given code snippet. Contracts and interfaces are similar to classes and interfaces in object-oriented programming (OOP) languages, respectively. A contract may inherit one or more parent contracts or interfaces. It consists of state variable declarations, functions, and function modifiers. While state variables and functions are comparable to instance variables and methods in OOP, they have Solidity-specific features. In the given code snippet, stateAddress is an internal state variable whose type is address. foo is a payable external function with a modifier called funcModifier. A function or function modifier is composed of a sequence of statements which may use various expressions as in OOP methods.

Solidity’s programming constructs in a program file are designed into four levels:

- Inter-module: this level involves signatures of modules and relationships among them. For example, inheritance is an inter-module level construct.
- Intra-module: this level involves the immediate constructs within a module. For example, intra-contract level includes state variable declarations, signatures of functions, and signatures of function modifiers.
- Intra-function and function modifier: this level involves individual statements within a function (excluding functions in interfaces)
- Intra-statement: this level involves components (e.g., expression) within a statement (e.g., function call of assignments)

```
pragma solidity 0.4.24 //version of Solidity to use

contract A is IA{

 address internal stateAddress; //internal state variable of the address type

 function foo() funcModifier payable external {
 int memory memInt; //int variable with data location of memory
 stateAddress = 0x12345; //address being assigned
 msg.sender.send(10); //sends ether to the contract
 msg.sender.call(0x123); //sends the bytes 0x123 to the contract
 selfdestruct(stateAddress); //self-destruct and send remaining ether
 ...
 }

 modifier funcModifier() {
 if (msg.sender == stateAddress) {
 _; //execute the function normally if true
 }
 }

}

library mathLib {...}

interface IA {...}
```

For each level, we identify feasible faults with respect to the programming constructs. For example, an incorrect use of inheritance is an inter-module fault (e.g., "is IA" is removed from "contract A is IA"). Missing the reference and definition of a function modifier in a contract is an intra-contract fault (e.g., funcModifier is removed from the signature of foo and the definition of funcModifier is removed). Missing a statement within a function of a contract is an intra-function-level fault (e.g., msg.sender.send(10) is removed from the function foo), whereas an incorrect expression within a statement is an intra-statement level fault (e.g., 10 is changed to 1 in msg.sender.send(10). The above classification ensures that the fault model covers the fault types of every programming construct in Solidity.

3.4 Mutation Operators

This section details and highlights the mutation operators that target Solidity-specific features. The mutation operators are categorized by the operator's associated programming construct level as was defined in our fault model. Deviant also includes general mutation operators, meaning that these mutation operators exist in other mutation testing tools and the programming constructs are common in other programming languages. These general mutation operators are listed and explained in Appendix A.

3.4.1 Intra-Statement Level

Intra-statement level mutation operators are those that modify the components within a statement. Typically, intra-statement level operators modify different types of expressions. In regards to Solidity-specific features, these mutation operators modify Solidity constructs such as: gas, address, address function, and data location constructs.

Gas Operators: The only mutation operators that take place on gas involve the modification of the literal value that is associated with the gas stipend. Deviant modifies the literal value to either a zero or random non-zero value. If the contract's gas value is modified, then it may cause the execution to stop either prematurely or continue on too long. As noted previously, the out-of-gas exception is a problem that occurs in Solidity. While this exception does not directly relate to the send function call problems that can cause an unexpected out-of-gas exception, an insufficient gas stipend can cause an out of gas exception for developers that don't understand how much gas they should actually allocate. The gas mutation operators are Modify Function Gas Value to Non-Zero (FGVNZ) and Modify Function Gas Value to Zero (FGVZ).

Address Operators: Address operators mutate attributes of an address variable. Addresses in Solidity can be represented as any numerical value. However, they are typically represented in a hexadecimal format. The modify address operators work similarly to other numerical literal operators. They modify the actual value that the address is being assigned to. These mutation operators can replicate faults such as type cast error, call to the unknown, and ether lost in transfer. The type cast error and call to the unknown can occur because the address can potentially become associated wrong contract. This will mostly only occur however if the address still points to a valid address that contains contract code. The ether lost in transfer is much simpler fault that can occur from this mutation operator, considering that either a random or zero address will most likely point to an orphaned address. The Switch Call Expression Casting (SCEC) mutation operator is only ever applied if there are two or more instances where addresses are being cast to different contracts. This mutation operator is more likely to cause type cast and call to the unknown faults because the addresses in the mutant contract should still be valid. These faults can be replicated because the SCEC operator takes two instances of addresses being casted to different contracts and then switches the contracts that they are being cast to, most likely causing them to point to incorrect addresses. Table 3.1 lists the address operators.

Table 3.1 Address Mutation Operators

Mutation Operators	Abbreviations	Fault Types
Switch Call Expression Casting	SCEC	Call to the Unknown Type Casts Ether Lost in Transfer
Modify Address Literal to Non-Zero	MALNZ	Call to the Unknown Ether Lost in Transfer
Modify Address Literal to Zero	MALZ	Call to the Unknown Ether Lost in Transfer

Address Function Operators: There are several functions that are associated with addresses, including: transfer, send, and call. In any of these cases, there is likely to be a literal associated with the function call. It is relatively simple to modify these literal values to any other value to create a valid mutant. These mutation operators modify the literal argument values that are in the function call, which typically represent the amount of ether that is sent. A potentially more dangerous fault that could exist in a smart contract is using the incorrect address function. For instance, both the transfer and send function calls have a gas stipend of 2,300, but differ in how they handle failure. The send function call returns a boolean false on failure, while the transfer function call throws an exception on failure. Even more problematic is the call function call, which does not have the same gas stipend of 2,300. When using call.value() in place of send or transfer to send ether, gas will continue to be spent until all of the user’s gas has been depleted. This means that it is highly likely that the call function can use more than 2,300 units of gas if the code execution continues on that long. Call is potentially the most dangerous out of all the function calls because it is not considered safe against reentrancy attacks, such as the DAO attack that was explained earlier. By swapping these function calls we can introduce faults such as reentrancy, gasless send, or exception disorder. Reentrancy can occur if a mutation operator replaces a send or transfer with a call function call. A gasless send can occur if a mutation operator can occur any time that any of the address functions are replaced by a send function call, but it will most typically happen when a transfer call is replaced with send. An exception disorder fault will typically occur when transfer or send are swapped. This fault will happen because if the developer was originally using transfer, then they will not be checking to if the call is returning false, while the mutant version will return false instead of throwing an exception like it normally would have in the code. Table 3.2 lists the mutation operators that target address functions.

Table 3.2 Address Function Mutation Operators

Mutation Operators	Abbreviations	Fault Types
Modify Transfer Value	MDTV	Incorrect Transaction Value
Modify Send Value	MDSV	Incorrect Transaction Value
Modify Call Value	MDVC	Incorrect Transaction Value
Transfer -> [Send, Call]	TRS, TRC	Out-of-Gas Exception Reentrancy Exception Disorder
Send -> [Transfer, Call]	STR, SC	Reentrancy Exception Disorder
Call -> [Transfer, Send]	CTR, CS	Out-of-Gas Exception Exception Disorder

Data Location Operators: The data location mutation operators swap the memory location keyword in the source code. The purpose of these mutation operators is to affect the behavior of the variables. For instance, storage variables persist beyond the lifespan of a function call, whereas the life span of a variable with the memory keyword is temporary, only existing in the function that it is declared in. Also, by design storage is quite a bit more expensive to use than memory, but this allows for storage variables to be more dynamic. This means that data types such as arrays are automatically assigned to the storage location to allow for dynamic usage. It is also important to note that memory variables cannot exist outside of the lifespan of a function, meaning that global variables cannot be declared as memory variables. Now, while these mutation operators cannot necessarily be directly associated with the faults that were listed in the fault taxonomy, it is quite clear that these mutation operators can severely affect the behavior of the variables and cause unexpected behavior. The Data Location Operators are change Storage to Memory (STRME) and change Memory to Storage (MESTR).

3.4.2 Intra-Function Level

Function level mutation deals with the mutation of the contents within a function. These contents include function calls, statements, or blocks, which may be comprised of multiple function calls or statements.

Event Operators: Events can fire from the smart contracts and anything connected to the Ethereum JSON-RPC API can listen to the events and then act. In the context of Solidity, events are primarily used for the EVM logging facilities. When an event is called, the arguments are stored in the transaction log, which resides in the blockchain. In dApps, when events are fired, the JavaScript can be notified and can then act accordingly. Of course, by applying these mutation operators, the dApp will potentially act improperly because an event was fired when it should not have or may not fire at all. These mutation operators will typically introduce an exception disorder fault because of the change of behavior that the mutation operators will introduce. If the developer does not properly test their event invocations, then it is likely that these mutants may be live when the mutation testing takes place. The event operators are Remove Event Invocation (REI) and Swap Event Invocations (SEI).

Selfdestruct Operators: Selfdestruct operators are essential for creating gas efficient contracts and for when a contract is at the end of its life. Calls such as selfdestruct (suicide is an alias to selfdestruct) send the entirety of a contract’s balance to the address that was defined in the argument and are then subsequently destroyed at its address. This features primarily exists to allow for the developers to destroy its own contract when they are done with it. This is incredibly useful because of the immutability attribute that the blockchain has, meaning that if a bug or unwanted behavior exists in the contract, then the developer can destroy that contract and redeploy a different one. It is also important in terms of gas because it technically costs negative gas to execute this function because it ultimately ends up freeing gas on the blockchain. Of course, calling these functions unexpectedly can have major consequences, considering that it deletes the contract code that resides at a specific address. For instance, take the example of someone trying to send ether to this address. If the selfdestruct call has been executed unexpectedly at this address, then the ether will be sent to an orphaned address, causing the ether lost in transfer fault. Table 3.3 lists the selfdestruct operators.

Table 3.3 Selfdestruct Mutation Operators

Mutation Operators	Abbreviations	Fault Types
Remove Selfdestruct Call	RSDC	Exception Disorder
Insert Selfdestruct Call	ISDC	Ether Lost in Transfer Exception Disorder
Remove Suicide Call	RSC	Exception Disorder
Insert Suicide Call	ISC	Ether Lost in Transfer Exception Disorder

Exception-Handling Operators: Exception-handling is a serious problem that exists in the EVM. As discussed earlier, exception disorder is a fault for Solidity because the EVM does not handle all errors the same. Because of the nature of error-handling in Solidity, exception-handling operators provide a lot of insight into a test suite’s ability to handle exceptions. A good example of the exception ambiguity exists in the difference between the send and transfer calls for sending ether to a contract. As explained earlier, send only returns false on the failure, while transfer throws an exception on failure. In some instances, a programmer might include a require statement on a send function call, this is actually equivalent to the transfer function call. In terms of mutation, if this require statement is removed, then an error will not be thrown and cannot be caught by the contract that has that code. This is a subtle error, but if the exception-handling is not done properly, then the code may still execute because it does not catch the exception that has occurred. Also, the revert statement can cause major problems for developers if it used improperly. The importance of revert cannot be understated in contract development. When something goes wrong in the execution of the contract code, it may be important to revert all state changes that happened during the execution, which is where the revert call comes in. The revert call reverses all the state changes, refunding all the remaining gas to the caller. Now, if the revert statement is in the wrong place or does not even exist, then these changes will probably not be reversed, causing some of the changes to be permanent. The exception-handling operators target these problems by intentionally removing or inserting these statements into areas of the code where they did not previously reside. Table 3.4 lists the exception-handling operators.

Table 3.4 Exception-Handling Mutation Operators

Mutation Operators	Abbreviations	Fault Types
Remove Require Statement	RRQS	Exception Disorder
Insert Require Statement	IRQS	Exception Disorder
Remove Assert Statement	RAS	Exception Disorder
Insert Assert Statement	IAS	Exception Disorder
Remove Revert Statement	RRVS	Exception Disorder
Insert Revert Statement	IRVS	Exception Disorder

Change Function Modifier Condition (CFMC) Operator: This mutation operator specifically targets conditions that exist inside function modifiers. Function modifiers are attached to functions, where the function modifier will always execute before the actual function that it is attached to does. Typically, they can be used to check preconditions before the function execution. Now, this mutation operator modifies the conditions that exist inside the modifier body. Specifically, the mutation operator moves the underscore character such that it will always execute, regardless of the conditions that exist in the modifier. This mutation operator can cause the program to have unexpected behavior because the state of the application will not be properly checked inside the function modifier.

3.4.3 Intra-Module Level

State Variable Visibility Operators: These mutation operators modify the visibility of state variables in Solidity. Unlike functions in Solidity, state variables do not allow the external keyword. Public state variables has a generated getter function generated when it is called. They can also be called by the internal contract or through messages. Internal variables are similar to internal functions in that they can only be accessed internally or through child contracts. Private variables are also similar to private functions in that they are only visible to the internal contract and cannot be accessed by child contracts. Visibility operators exist in many languages, however, with the nature of Solidity, it is important to distinguish them from other operators, given the nature of the EVM. For example, with the inclusion of the internal keyword, developers may be confused between the semantic similarity that exists with the private keyword. Intuitively, the two words may seem as if they are just an alias for each other. However, the Solidity documentation clearly outlines that there is a semantic difference between the two visibilities. If a developer is to confuse the two visibilities, it is likely that the variable can be accessed in an unintentional manner. Table 3.5 lists the state variable visibility mutation operators.

Table 3.5 State Variable Mutation Operators

Mutation Operators	Abbreviations	Fault Types
Public → [Private, Internal]	PUPI, PUI	Loss of Visibility
Private → [Public, Internal]	PRPU, PRI	Gain Excess Visibility
Internal → [Public, Private]	IPU, IPR	Loss of Visibility Gain Excess Visibility

Function Type Operators: The three primary function types in Solidity are pure, view, and payable. Pure and view functions deal with how the function is allowed to affect the state of the program. Pure functions are ultimately a subset of view functions considering that they are essentially more restrictive in what they can access in terms of the state of the program. Payable functions have no relation to the other two function type keywords; however, they do serve an important purpose in Solidity. By swapping or removing the pure/view keywords, the developer may be able to realize if they are using the keywords incorrectly. They may also notice that a mutation may be live for either of these types of mutations, probably indicating that they are only considering the happy-path scenario for testing this function. The payable function mutation is relatively self-explanatory, as if a test doesn’t recognize that a payable function has been modified, they probably aren’t actually trying to send Ether to the function, which of course it the primary purpose of the modifier keyword. Table 3.6 lists the function type mutation operators.

Table 3.6 Function Type Mutation Operators

Mutation Operators	Abbreviations	Fault Types
Pure → View	PUV	Gain Readability of State
View → Pure	VPU	Lose Readability of State
Delete Pure Keyword	DPUK	Gain Control of Program State
Delete View Keyword	DVK	Gain Control of Program State
Delete Payable Keyword	DPAK	Function Cannot Receive Ether
Insert Payable Keyword	IPAK	Function Can Receive Ether

Function Visibility Operators: There are four types of function visibility in Solidity: public, private, internal, and external. These mutation operators swap the different function visibility keywords around. These mutation operators are useful because of the major differences that exist between the different types of visibility. For instance, external functions can only be called by other contracts and transactions, while internal functions can be called by the function it is declared in and all contracts that are derived from it. Now, while public and private might seem semantically similar to external and internal functions, they are not. Public functions allow for anyone to access, this is different from external, where only outside transactions and functions can call it. On the other hand, private functions can only be called from the contract that it is declared in, meaning that no derived contracts can access this function. In many cases, the mutation results that come out of these mutation operators are similar to those of the state variable visibility operators. What is meant by this is that the mutation operators will also give similar insight as to how the functions are being tested when compared to the state variables. Table 3.7 lists the Function Visibility operators.

Table 3.7 Function Visibility Mutation Operators

Mutation Operators	Abbreviations	Fault Types
Public → [Private, Internal, External]	VPUPR, VPUI, VPUE	Gain Visibility Lose Visibility
Private → [Public, Internal, External]	VPRPU, VPRI, VPRE	Gain Visibility Lose Visibility
Internal → [Public, Private, External]	VIPU, VIPR, VIE	Gain Visibility Lose Visibility
External → [Public, Private, Internal]	VEPU, VEPR, VEI	Gain Visibility Lose Visibility

Modifier Signature Operators: The function modifiers that exist in Solidity are used to execute code before a function is called. Primarily, these functions are used to check a precondition before the actual function is ran. Now, while these operators are associated with function modifiers, they are not the same as the Chance Function Modifier Condition operator that was explained earlier. Instead, these mutation operators modify the modifiers that are attached to function signatures that exist in a Solidity smart contract. This means that certain conditions will be checked that might not necessarily have anything to do with the actual function that is being executed. In many cases, this may result in the function not executing or if a function modifier is removed, it could potentially always run. Because of the unexpected behavior that this change will introduce, there is a high probability for exception disorder to occur because of the unchecked conditions that will ultimately take place because of the mutations.

- *Delete Function Modifier (DFM):* This mutation operators deletes any modifier argument associated with a function. This should cause the precondition check to be ignored when the contract is running.
- *Insert Modifier on Function (IMF):* In the case that there is no modifier argument attached to a function, while a modifier still exists in a contract, a mutation can occur where a modifier argument is attached to a function.

Library Function Visibility Operators: Libraries are an important part of the EVM. Libraries serve a different purpose than contracts in that they are deployed once to a specific address and then their code is reused by contracts that use the libraries. Again, libraries contain functions with visibility modifiers, however, they function slightly different than normal contracts. For instance, the internal keyword causes the function to be inclined into the calling contract's bytecode. The reason that this exists is that for some smaller libraries, it is more efficient to compile the function inline rather than to link the bytecode. These mutation operators are for the most part the same as the function visibility mutation operators; however, we only include the visibility keywords that exist for Solidity libraries. These mutation operators should ultimately give similar insight when compared to the function visibility operators. Table 3.8 lists Library Function Visibility operators.

Table 3.8 Library Function Visibility Operators

Mutation Operators	Abbreviations	Fault Types
Public → [Private, Internal]	LPUPR, LPUI	Gain Visibility Lose Visibility
Private → [Public, Internal]	LPRPU, LPRI	Gain Visibility Lose Visibility
Internal → [Public, Private]	LIPU, LIPR	Gain Visibility Lose Visibility

3.4.4 Inter-Module Level

Currently, the only mutation operator that targets the inter-module level of a Solidity smart contract deals with multiple-inheritance. Unlike Java, Solidity supports multiple-inheritance. There are many complicated problems that can potentially exist with a contract that inherits multiple parents. One of the most famous problems that exist is the diamond problem [19], where classes B and C inherit from A and then D inherits from B and C. The reason that this is a problem is because of the same function in A is overridden in B and C, then which version of the function will D use? In the case of Solidity, this problem is solved by using C3 linearization [20], which is also the method that Python uses for solving its diamond inheritance problem. Currently, the only operator that covers multiple-inheritance is the Remove One Parent (ROP) operator, which removes a parent from its inheritance structure. Now, this operator may cause many potential problems. If there exist two parents in the inheritance structure which share the same function signature, then the remaining parent after the mutation will haves its function used instead of the other parent. This could result in the function remaining functionally equivalent, though it is likely that the entire program will still not be functionally equivalent. On the other hand, the function could operate quite differently.

Determining the behavior is relatively difficult as of now but could be a potential area of interest later on.

3.4.5 Mutation Operator Relationship to Fault Types

The fault types provided by Atzei et al. [3] and discussed in Chapter Two can be replicated by the some of the mutation operators that were designed in Deviant. The fault types we consider are those that can observed within Solidity code itself, namely: call to the unknown, exception disorder, gasless send, type casts, reentrancy, and ether lost in transfer. Table 3.9 lists these fault types along with the associated Solidity-specific mutation operator category that can replicate these faults.

Table 3.9 Fault Types with Associated Mutation Operators

Fault Type	Mutation Operator Category
Call to the unknown	Address
Exception disorder	Address Function
	Selfdestruct
	Exception-Handling
Gasless send	Address Function
Type casts	Addresses
Reentrancy	Address Function
Ether lost in transfer	Address
	Selfdestruct

Call to the unknown: Address mutation operators can potentially create call to the unknown faults in Solidity. For example, if the mutated literal value that is assigned to an address is then cast and used as a contract, then the address will no longer be correctly pointing to the location where the contract is stored on the blockchain. If this is the case, then when an attempt to use that contract is made, there will either be no contract at that location or the contract that exists at that location will have its fallback function invoked. Essentially, the developer will not be able to anticipate what will happen when that mutated address is used.

Exception disorder: There are three Solidity-specific mutation operator categories that can cause an exception disorder to occur: address function, selfdestruct, and exception-handling. Address function operators can cause an exception disorder to occur typically when modifying between the send and transfer function calls. This is because send returns false on failure, whereas transfer throws an exception. This means that if there is no exception checking done on a send call, then the exception might not be caught and the user of the Solidity application might not be aware that something went wrong. The selfdestruct operators can either insert a selfdestruct statement or remove a selfdestruct statement. In either case, it will cause the contract to either exist or not exist when it shouldn't. When this happens, a contract may attempt to use the contract that has been incorrectly destroyed from the blockchain, causing an exception that might not make any sense to the developer (e.g. reverting the transaction suddenly). Exception-handling operators either insert or remove exception-handling statements in the Solidity smart contracts. By doing this, exceptions can either occur or not occur in the inappropriate place. When this happens, improper exception-handling can cause the program to behave unexpectedly. Also, by introducing exceptions in certain locations, then gas may not be refunded.

Gasless send: Address function operators, specifically modifying transfer to send, can cause a gasless send to occur. This is because when the send is compiled to EVM bytecode, it is compiled to a call function with a gas stipend of 2,300. Now, since the call function will have no signature, it will invoke the fallback function. If the fallback function of the contract has too many executable steps, then the gas that was allocated for the send will run out, causing an out-of-gas exception. This means that if transfer was able to work around this error, then by using send instead, it will cause an out-of-gas exception to occur when it should not have.

Type casts: Type cast faults occur with the address mutation operators and are similar to the call to the unknown faults. When a mutation operator such as the Switch Call Expression Casting (SCEC) operator is applied, it takes two instances of addresses being cast to a contract and then swaps them. When this occurs, it is highly likely that it will cause a type cast error. This is because the Solidity compiler cannot check to see if the address contains the actual contract that it is being cast to, rather it can only check the interface of the contract itself. By swapping these contract casts, the two contracts now point to the incorrect contract, causing the contract to have its fallback function invoked when it is called.

Reentrancy: Reentrancy vulnerabilities can occur with address function operators, more specifically, they can occur when either transfer or send is replaced with the call function. The call function itself is not safe from reentrancy attacks, especially in instances where ether is being sent. This is because unlike transfer or send, call does not have an explicit gas stipend, rather a Solidity developer needs to include the gas stipend themselves. When call is used instead of send or replace, code will continue executing until either the gas runs out for the contract, or there are no more executable steps. In the case that there is an attacker, an attacker would just simply recall the function where the vulnerability exists. This is because when call is used to send ether to a contract, it will invoke the receiver's fallback function, where an attacker may recall the vulnerable function where call was used. If this happens, then the ether transaction will be repeated until the gas runs out. Now, this will cause an exception. However, because call is being used, it will only revert the last transaction that took place, meaning that all the other transactions that took place before it will be considered valid. What this means is that when call replaces send or transfer, it can potentially introduce a reentrancy vulnerability that allows the vulnerable contract’s non-recursive function to be entered recursively.

Ether lost in transfer: Ether lost in transfer faults can happen when an address or selfdestruct mutation operator is applied. When an address mutation operator modifies the literal value that is associated with an address, that address will likely point to an address that contains nothing at that location. When a contract attempts to send ether to this address, it will cause an ether lost in transfer fault. This is because that ether will be sent to an orphaned address (nothing exists at that address) and the ether will not be recoverable. When a selfdestruct insert mutation operator is applied, it will cause the contract to be destroyed when it is executed. When this happens, any time that a contract attempts to send ether to the address where to contract used to be, it will end up being lost. This is because that address is no longer valid as the contract can no longer be invoked at that address. The ether then becomes lost and is unrecoverable.

CHAPTER FOUR: IMPLEMENTATION

Chapter Four explains our implementation of Deviant. Section 4.1 details our implementation of the Deviant application, discussing both the Graphical User Interface (GUI) and explaining how we have implemented our architecture in relation to our design in Chapter 3. Section 4.2 addresses the implementation issues that were faced in Deviant.

4.1 Implementation of Deviant

Deviant provides an easy-to-use Graphical User Interface (GUI) for developers. The main purpose of the GUI is to provide organized and meaningful information to the user in an intuitive manner. The GUI itself is implemented using Electron, which is a NodeJS framework that allows for developers to create GUIs in a similar manner to web pages. Much of the styling is done using bootstrap, considering that bootstrap is widely used and removes much of the hassle that is required to manually style the HTML pages.

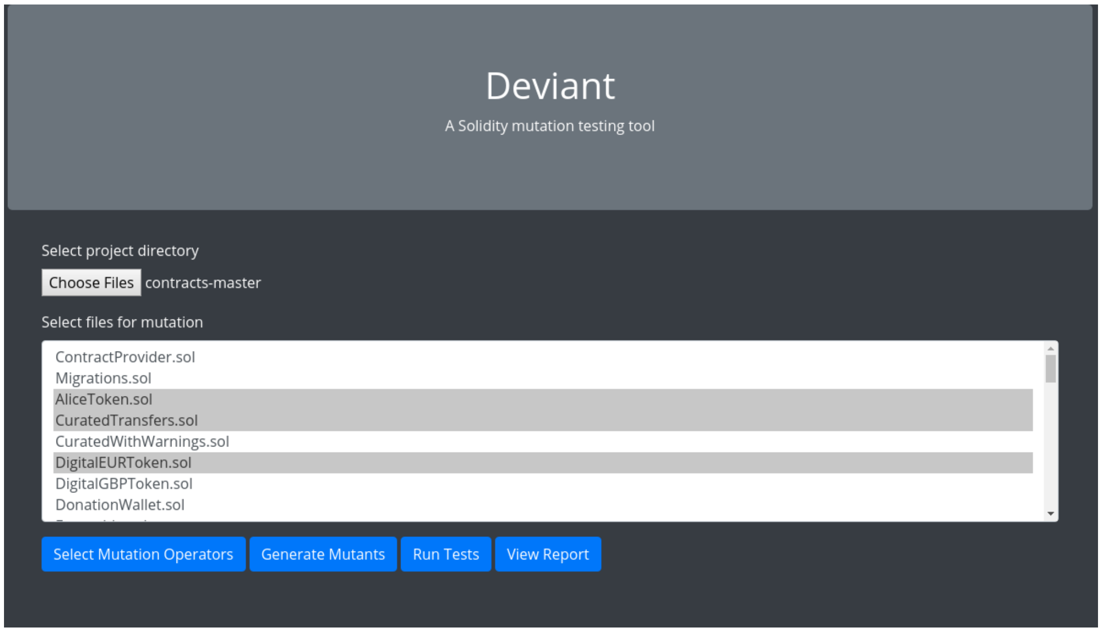
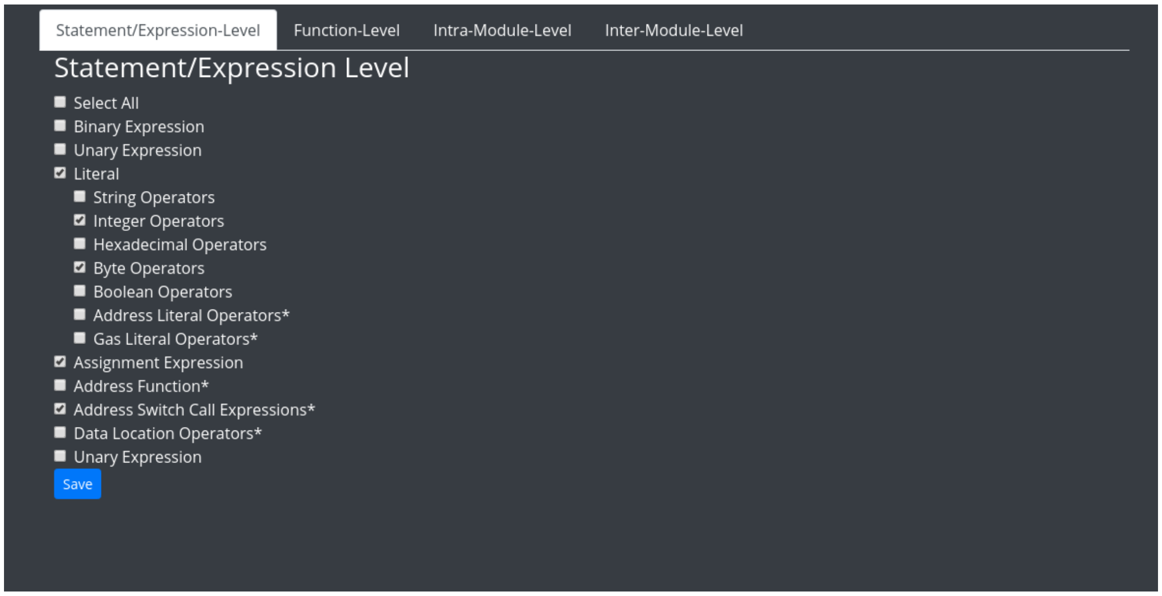


Figure 4.1 Home Page for Deviant GUI

As seen in Figure 4.1, the Deviant GUI allows the user to select the relevant Solidity project directory for mutation. After selecting the project, the user can then choose which Solidity files they wish to use for mutation. On the homepage, we include four buttons: Select Mutation Operators, Generate Mutants, Run Tests, and View Report. The user first must select the mutation operators as seen in Figure 4.2, which afterwards they can then generate the mutants. After the mutants have been generated, the user can then proceed to the run the tests. Finally, once the tests have finished executing, the user can then view the report.

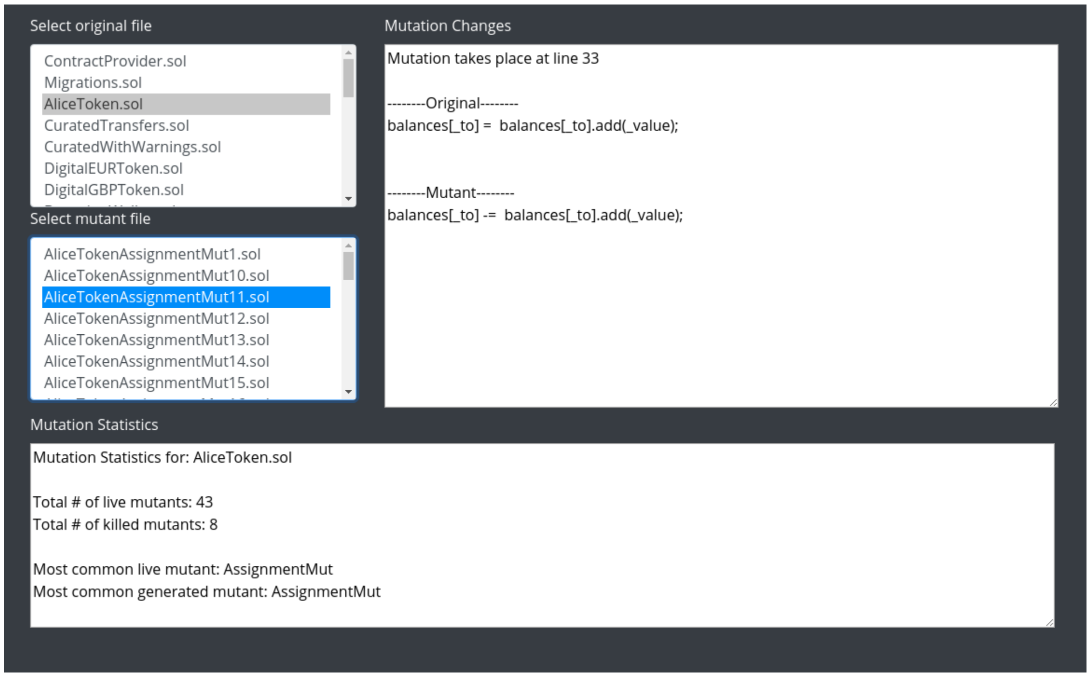


Statement/Expression Level

- [ ] Select All
- [ ] Binary Expression
- [ ] Unary Expression
- [✓] Literal
  - [ ] String Operators
  - [✓] Integer Operators
  - [ ] Hexadecimal Operators
  - [✓] Byte Operators
  - [ ] Boolean Operators
  - [ ] Address Literal Operators*
  - [ ] Gas Literal Operators*
- [✓] Assignment Expression
- [ ] Address Function*
- [✓] Address Switch Call Expressions*
- [ ] Data Location Operators*
- [ ] Unary Expression

Figure 4.2 Mutation Operators Page for Deviant GUI

The mutant operators in the GUI are divided into their respective categories: statement/expression level, function level, intra-module level, and inter-module level as seen in Figure 4.2. We have divided the mutation operators into their respective categories to allow the developers to consider which mutation operators they find most important during their own evaluation. In most instances, a developer does not need to include every mutation operator that is available, rather they only want to include a select few that are of interest to them.


Select original file
ContractProvider.sol
Migrations.sol
AliceToken.sol
CuratedTransfers.sol
CuratedWithWarnings.sol
DigitalEURToken.sol
DigitalGBPToken.sol
Select mutant file
AliceTokenAssignmentMut1.sol
AliceTokenAssignmentMut10.sol
AliceTokenAssignmentMut11.sol
AliceTokenAssignmentMut12.sol
AliceTokenAssignmentMut13.sol
AliceTokenAssignmentMut14.sol
AliceTokenAssignmentMut15.sol
Mutation Changes
Mutation Statistics

```
Mutation takes place at line 33

--------Original--------
balances[_to] =  balances[_to].add(_value);


--------Mutant--------
balances[_to] -=  balances[_to].add(_value);
```

```
Mutation Statistics for: AliceToken.sol

Total # of live mutants: 43
Total # of killed mutants: 8

Most common live mutant: AssignmentMut
Most common generated mutant: AssignmentMut
```

Figure 4.3 Report Page for Deviant GUI

The report page, as seen in Figure 4.3, allows the developer to select the original file from the project that was selected, along with a mutant file that they wish to compare to the original. The report will display the line where the mutation occurred, along with the line number in code where the mutation took place. The bottom section of the report displays the total number of mutants for the selected file, along with the number of killed mutants, live mutants, and mutation score. The report also includes the most common live mutant type along with the most common generated mutant type. It should be noted that the report page does not automatically determine if a mutant is functionally equivalent to the original program. Deviant cannot automatically determine equivalent mutants, instead the developer must evaluate the generated mutant and determine if the mutant is equivalent to their program. This means that the number of live mutants reported on the report page may be incorrect with relation to the equivalent mutants.

Deviant is implemented using JavaScript with NodeJS. JavaScript was selected as the primary language for two reasons: the available libraries that exist for the modification of Solidity source code and the overall support that exists for NodeJS in regards to Solidity. In fact, most of the dApps that are written with Solidity also use a NodeJS portion for things such as its front-end or event-handling. Using NodeJS is also incredibly useful for this project, considering that the Solidity language itself is heavily influenced by JavaScript and shared much of the same syntax and other features.

Beyond the programming language, Deviant primarily uses two libraries to modify the Solidity source code, solparse [13] and solmeister [14]. Solparse takes in the Solidity code as input, parses the code, and then outputs an AST (Abstract Syntax Tree) based on the Solidity contract. This AST is represented in the JSON (JavaScript Object Notation) format as seen in the below JSON snippet, which allows for easy traversal and is fairly straightforward to modify. Deviant then uses solmeister to traverse the AST, looking for tree nodes that are relevant to mutation operators. When a relevant node is found, then the mutation operator is applied, and a new mutant AST is created. In this case, there exists only one change per mutant. After the mutant AST has been created, solmeister has the ability to generate source code based off the mutant AST. The newly generated source code is then written to file and stored in a mutant project directory that contains all other mutants for the project. Referring again to the JSON snippet below, we can see an individual node that is part of a Solidity smart contract. In this example, highlighted in red, Deviant would modify the operator value in the node from ‘<’ to another relational operator such as ‘>’ ,’ >=’, ‘<=’, ‘==’, or ‘!=’.

```
"type": "IfStatement",
"test": {
"type": "BinaryExpression",
"operator": "<",
"left": {
"type": "MemberExpression",
"object": {
"type": "Identifier",
"name": "balances",
"start": 601,
"end": 609
},
"property": {
"type": "MemberExpression",
"object": {
"type": "Identifier",
"name": "msg",
"start": 610,
"end": 613
},
"property": {
"type": "Identifier",
"name": "sender",
"start": 614,
"end": 620
},
"computed": false,
"start": 610,
"end": 620
},
"computed": true,
"start": 601,
"end": 621
},
"right": {
"type": "Identifier",
"name": "amount",
"start": 624,
"end": 630
},
"start": 601,
"end": 630
}
```

After all of the mutants have been generated, Deviant then runs the test suites against the mutant versions of the Solidity program files. Truffle is a popular development environment, testing framework, and asset pipeline for Ethereum [15]. Deviant requires that the provided Solidity projects are developed using the Truffle environment, which for most Solidity projects, is already the case. The reason that Deviant requires Truffle is primarily for automation purposes, without Truffle, the process of mutation testing would be quite cumbersome. Truffle provides a systematic approach to testing smart contracts, while also being the most popular development environment for Solidity. What Deviant actually does with the Truffle project is to actually run the test command for Truffle, while making sure that the appropriate mutant is being compiled instead of the original source code.

Truffle's testing phase always starts out with the contracts being recompiled. In our case, we insert the mutant into the project's contract directory and ensure that the mutant is compiled while the original source code is ignored. This process is repeated until every mutant has been ran against the entirety of the test suite, meaning that only mutant is used at a time. Deviant tracks these mutants and then reports the evaluation metrics to the user.

4.2 Implementation Issues

There are many implementation challenges that occur during the development process of a mutation tool for a relatively new programming language. Many of these implementation challenges relate closely to the design challenges that were mentioned in the overview subsection of Chapter 3. In relation to the constant changes that take place to the Solidity language itself, developers have to be aware of the changes and how they affect the current implementation of their program, ours included. The most noticeable changes in our development process was the syntactic and semantic differences that existed between the different Solidity versions. As explained in Chapter 3, until Solidity 4.24, the constructor in the contract was implemented by using the name of the contract as the function name, whereas it is now defined by using the constructor keyword. In terms of implementation challenges, if a developer were using an earlier version of Solidity, then their old contracts will no longer be compiled by the newer version of the Solidity compiler. To fix this, developers have to ensure that their implementation follows the appropriate syntactic rules in relation to the version of the Solidity compiler that they are using. In our case, this means that we only support Solidity versions that are great than 0.4.0, which for almost all projects should not be an issue. While this individual change may be small, there are numerous changes that occur between the different Solidity versions.

In any mutation testing tool, there may be a possibility to generate an invalid mutant. An invalid mutant is a mutant that does not compile. In our implementation, there are several instances where an invalid mutant may be generated. This issue is related directly to the design challenge mentioned in Chapter Three about the generate of invalid mutants. If we do not consider this problem, then the mutation testing performance will be severely hindered. This is because for every mutant that is generated and considered invalid, will still have to fail its compilation, ultimately costing time. To bypass this problem, we implement extensive precondition checking of the AST according to the Solidity documentation. This means that every time that we encounter a potential node, we have to evaluate several attributes that may exist in the node. If these attributes are not what we expect, then we do not create the mutation at that point. However, there are still instances where an invalid mutant could be generated. In our application, we have implemented a check that looks for compile-time errors and if one exists, we consider that mutant to be invalid and it is no longer considered in our report.

The constant updates to Solidity not only affect the syntax and semantics of the programming language, but they also affect many of the libraries that are useful for our mutation tool. For instance, the solmeister library that we use has not been updated since 2016, having several new versions of Solidity come out since then. To overcome this problem, we had to modify the project’s own dependencies to ensure that its own libraries are updated to work with the appropriate version of Solidity. It is quite fortunate that the fix was this simple, but for most libraries this will not be an appropriate solution. In many cases, abandoned libraries might have to be retooled to work for the developer that is using them or they may have to give up on the library completely. Thankfully, our situation relies on libraries that are not overly-complicated and can be easily fixed in the future if we need to.

CHAPTER FIVE: EMPIRICAL EVALUATION

In this chapter we detail our empirical evaluation. Section 5.1 gives an overview of our experiment, providing details and metrics about our subject program along with providing the results from our experiment. Section 5.2 analyzes the results from our experiment.

5.1 Experiment

To evaluate the effectiveness of Deviant, we used the following Solidity applications that cover the main Solidity features:

- *MetaCoin* [22] : MetaCoin is part of a popular collection of Solidity smart contracts in the Truffle Box repository. MetaCoin is a boilerplate for the creation of a coin in Solidity.
- MultiSig Wallet [23]: The MultiSig Wallet is a popular implementation of multi-signature wallets for Ethereum. Multi-signature wallets are primarily used because they require multiple parties to sign before transactions are executed.
- *Alice* [21]*:* Alice is a social impact platform build on top of Ethereum. The unique aspect of Alice is that these social projects are ran transparently. The goal of this project is to allow organizations to identify and scale projects according to their performance.
- *AragonOS* [24] *:* aragonOS is a framework that can be used to develop dApps, protocols, and decentralized organizations. Specifically, aragonOS adds a layer of

abstraction for managing resources when creating decentralized organizations and protocols.

- *OpenZeppelin* [25]: OpenZeppelin is one of the most popular Solidity libraries on GitHub for secure smart contract development. OpenZeppelin provides many implementations of common Solidity components and standards that are used by the Solidity developer community.

Table 5.1 details the metrics of the subject program. All of them have much more test code than the production code. The tests for MetaCoin have achieved 100% and 50% of statement coverage and branch coverage of the production, respectively. For MultiSig Wallet, the statement and branch coverage reached 100%. The tests for Alice have not reached full statement coverage mostly because of exception handling code, which is not exercised unless the exceptions are triggers. The tests of aragonOS have reached almost complete statement and branch coverage. The tests of OpenZeppelin have achieved both 100% coverage for both statement and branch coverage based on coverage of relevant lines of code. The tests of MultiSig Wallet have achieved 100% statement and 100% branch coverage. We have retrieved the coverage statistics for aragonOS and OpenZeppelin from coveralls [42] as part of their continuous integration, while we retrieved the coverage results for Alice, MetaCoin, and MultiSig Wallet from solidity-coverage [43].

Table 5.1 Subject Program Metrics

Subject Program	Production Code (LOC)	# Contracts	Test Code (LOC)	# Tests	Statement Coverage	Branch Coverage
MetaCoin	29	2	71	4	100	50
MultiSig Wallet	501	7	673	9	100	100
Alice	1,040	22	1,852	129	86.25	59.09
aragonOS	2,492	43	3,840	549	99.79	98.85
OpenZeppelin	6,085	133	7,861	2188	100	100

The experiments were performed on several different machines. The first machine is a Lenovo T430 laptop, the second a Boise State provided workstation, and the third is a PC with 32 GB of RAM and an i7 6700k at 4.6 GHz. For each subject program, we first generate the mutants of the program, run the tests against all mutants, remove live equivalent mutants, and report mutation scores. In the experiments, both contracts in MetaCoin are mutated with all operators. For MultiSig Wallet, we have generated mutation operators for all 7 of the contracts, including both Solidity-specific and traditional mutation operators. Of the 22 contracts in Alice, 19 are meaningful and applicable. They are mutated by all operators. Due to the complexity of aragonOS and time-consuming of the mutation testing, currently we have applied mutation testing to 32 of the 43 (considering only the meaningful and applicable modules) contracts in aragonOS, only generating the Solidity-specific mutation operators. For reasons similar to aragonOS, we have only generated Solidity-specific mutants for OpenZeppelin, targeting 16 of the 133 Solidity files. In OpenZeppelin, we specifically targeted larger program files that contained many of the unique Solidity features that we aimed to target for our evaluation. Table 5.2 lists the subject programs and the contracts that were selected for mutation in our experiment.

Table 5.2 Subject Programs and Their Contracts Used for Mutation

Subject Program	Solidity Files
MetaCoin	MetaCoin.sol ConverLib.sol
MultiSig Wallet	Factory.sol MultiSigWallet.sol MultiSigWalletFactory.sol MultiSigWalletWithDailyLimit.sol MultiSigWalletWithDailyLimitFactory.sol TestCalls.sol TestToken.sol
Alice	AliceToken.sol Coupon.sol CuratedTransfers.sol CuratedWithWarnings.sol DigitalEURToken.sol DigitalGBPToken.sol DonationWallet.sol Escapable.sol FlexibleImpactLinker.sol MockValidation.sol MoratoriumTransfers.sol OffChainImpactLinker.sol OwnableWithRecovery.sol Privileged.sol Project.sol ProjectWithBonds.sol
aragonOS	ACL.sol ACLSyntaxSugar.sol APMNamehash.sol APMRegistry.sol AppProxyBase.sol AppProxyPinned.sol AppProxyUpgradeable.sol AppStorage.sol AragonApp.sol BaseEVMScriptExecutor.sol CallsScript.sol DAOFactory.sol ENSConstants.sol ENSSubdomainRegistrar.sol ERC20.sol EtherTokenConstant.sol EVMScriptRegistry.sol EVMScriptRunner.sol Initializable.sol Kernel.sol

	KernelConstants.sol Repo.sol SafeMath64.sol ScriptHelpers.sol TimeHelpers.sol Uint256Helpers.sol UnsafeAragonApp.sol VaultRecoverable.sol Autopetrified.sol DelegateProxy.sol Petrifiable.sol UnstructuredStorage.sol
OpenZeppelin	ECDSA.sol ERC165.sol ERC165Checker.sol ERC1820Implementer.sol ERC721.sol ERC777SenderRecipientMock.sol ERC20.sol Pausable.sol ReentrancyAttack.sol ReentrancyMock.sol PaymentSplitter.sol Roles.sol SignerRoleMock.sol StringsMock.sol WhitelistAdminRole.sol WhitelistedRole.sol

Table 5.3 shows the experimental results including the potential equivalent mutants. We were not able to manually check every mutant that was generated to determine if it was functionally equivalent to the original program, as doing so with such a large number of mutants would be incredibly time consuming. However, for the Solidity applications MetaCoin and MultiSig Wallet, we were able to manually check the generated mutants and determine which ones were considered equivalent. The mutation scores of Solidity-specific features of all subject programs are very low (36.36%-69.70%). This indicates that the existing tests are unable to reveal the majority of faults

Solidity-specific programming constructs although they have covered almost all statements (or even all branches in aragonOS and OpenZeppelin). As such, we believe that a test suite adequate for the statement and branch coverage of Solidity programs does not necessarily provide a high-level assurance of code quality.

Table 5.3 Subject Program Experimental Results Including Potential Equivalent Mutants

Subject Program	# Contracts Mutated	Mutation Method	Total Mutants	# Killed Mutants	Mutation Score
MetaCoin	2	All operators	44	29	65.91%
		Solidity-specific features	11	4	36.36%
MultiSigWallet	7	All operators	639	145	22.59%
		Solidity-specific features	220	128	58.18%
Alice	19	All operators	1,057	570	53.93%
		Solidity-specific features	431	165	38.28%
aragonOS	32	Solidity-specific features	793	355	44.77%
OpenZeppelin	22	Solidity-specific features	439	306	69.70%

We found that there were no equivalent mutants generated for the MetaCoin project, while the MultiSig Wallet project had 45 generated equivalent mutants. This led to MultiSig Wallet killing 145 out of all 594 mutants and killing128 out of 220 Solidity-specific mutants, receiving a Mutation score of 24.41% and 58.18% respectively. For these two subject programs, we noticed that removing the equivalent mutants did not affect the mutation score heavily.

5.2 Analysis

In our experimental evaluation, our subject programs had relatively low mutation scores. These low mutation scores indicate that the provided test suites are not very adequate at addressing common faults and vulnerabilities that exist in Solidity. We have also noticed that some mutant types are much more likely to be considered live than others. In Table 5.4, the subject programs are presented with the number of live mutants for each mutant type of Solidity-specific features. The total number of live mutants in Table 5.4 is equal to the number of total mutants minus the number of killed mutants from Table 5.3. According to our results, there are clearly some mutants that are killed less often than others. In this section, we discuss the mutant types in relation to our experiment and provide analysis as to why some of these live mutants are so prevalent.

Table 5.4 Number of Live Mutants for each Mutant Type of Solidity-Specific Features

	MetaCoin	MultiSig Wallet	Alice	aragonOS	OpenZeppelin
Gas	0	0	0	0	0
Address	0	0	0	0	0
Address Function	0	0	0	1	2
Data Location	0	0	0	3	0
Event	1	0	24	6	3
Selfdestruct	0	13	26	32	6
Exception-Handling	0	36	46	94	61
Modifier	0	13	50	10	10
Function Type	3	10	4	45	4
Function Visibility	2	0	67	109	0
Library Function Visibility	1	0	0	4	0
State Variable Visibility	0	20	49	134	43
Total	7	92	276	448	143

5.2.1 Gas Mutants

Considering that the gas operators only involved the mutation of the literal value that were passed to the gas call, these mutants were killed quite easily. As seen in our results, not a single gas mutant was marked as live in our experiment. Since the Ethereum platform introduced gas as a concept to limit the number of executable steps before the termination of a program, the mutations would just often times cause the program execution to terminate prematurely, causing exceptions to be propagated to other parts of the program quite quickly. In any case, the test cases that were provided were able to catch the exceptions that were thrown by the out-of-gas exceptions caused by the gas mutants.

5.2.2 Address Mutants

Address operators deal with the mutation of the address type. This either mutated the address literal or mutated the contract that the address was being cast to. In both cases, the mutants were always killed throughout our experiment. Modifying the address that are associated with a contract would have drastic effects on the program behavior. Considering that if a contract is pointing to the wrong address, when it is used, it will likely just invoke the fallback function. Now, if this happens, the program execution should break and the test suite will most likely catch this, as an unexpected exception should typically occur and propagate through the program. This means that each of the test suites that were evaluated were able to successfully kill the mutants that were generated.

5.2.3 Address Function Mutants

The number of generated address function mutants are not high in relation to some of the other mutants. This makes sense as send, transfer, or call functions are typically not prevalent throughout a Solidity project. Many of the address function mutants that were generated were marked as live during our experiment. In almost all instances, swapping between send, transfer, and call will not result in any major functional differences between the mutant and the original program version. However, the key difference as stated in the operator description is how each of these functions operate when an error-condition has been met. It is most likely the case that developers are not testing malicious or faulty scenarios when using transfer, send, or call. For instance, the major difference between send and transfer is that transfer throws an exception when failure has occurred while send returns false on failure. If failure never happens during the test suite, it makes sense as to why these mutants will not be detected. In our case, the reachability of this error-condition can be hard to achieve. In a test suite, it may not be feasible to attempt to cause a failure on send or transfer. For example, if attempting to get an out-of-gas exception while using send, it would require the developer to write extra and unnecessary code in their fallback function. This is not reasonable for most developers as extra code being written in the fallback function is considered a waste and costs extra gas. It is also important to note that in our experiments there are not many instances where Ether is being transferred. Most of our applications that were used are not heavy-financial applications, meaning that the instances of Ether transfer are fairly limited. Even if the applications were mostly financial, it wouldn’t make sense for there to be an abundance of transfer calls. Including too many instances of transfer or send calls would most likely cause the design of the application to be overly complicated or redundant. In any case, these mutants were typically almost never killed by the applications in our experiment even with the low number of generated mutants.

5.2.4 Data Location Mutants

The data location mutants were split based on if the mutant switched storage to memory or memory to storage. Storage to memory mutants were mostly considered live during our experiment. This is most likely due to the fact that the changes do not propagate heavily throughout the program. For instance, the original version of the program would have had the storage variable pointing to a state variable. If this is the case, when the assignment is made when the mutant is a memory variable, it will cause that change to just be localized to within the scope of the function. This would likely not cause any drastic changes to the program’s behavior. However, the memory to storage mutants were almost always killed. This is most likely because when a variable is declared as storage in the scope of a function, it will by default point to the state variables defined in the contract. Of course, if this value is overwritten because of this change, it will cause changes that propagate throughout the program. This means that there are several instances where the change can be caught by the test suite.

5.2.5 Event Mutants

Event mutants in most cases were killed, but in some situations were still live. Considering that events are typically used for EVM logging facilities and the JavaScript portion of the dApp it makes sense that sometimes events may be overlooked in a test suite. This is not necessarily just a problem in Solidity. In many cases, some events are triggered only in exceptional circumstances, meaning that the conditions for the events to be triggered cannot be easily determined by the developers. In these cases, it makes sense as to why these events would not be triggered, considering the difficulty of actually being able to target the event invocation. This means that in some cases, the event mutations are not reachable by the provided test suite. It can also be the case that the test suite does not correctly check the state of the program after the execution of the mutated statement. Take the following code snippet:

```
function investFromWallet(uint _amount) public {
 require(getToken().transferFrom(msg.sender, beneficiaryAddress, _amount));

 uint256 couponCount = _amount.div(couponNominalPrice);
 coupon.mint(msg.sender, couponCount);
 liability = liability.add(getPriceWithInterests(_amount));

 //emit CouponIssuedEvent(msg.sender, couponCount);
}
```

By removing the event invocation (commenting it out in the code), any event listeners will not catch that the event should have been invoked. When a test suite cannot detect this mutant, it suggests that it does not meet the necessity requirements, indicating that it does not check the state of the program immediately after the statement has been executed.

5.2.6 Selfdestruct Mutants

In our experiment, selfdestruct mutants were killed most of the time. However, there were more instances of live mutants than we were expecting. Now, we had originally assumed that selfdestruct mutants would almost always be killed, but there are actually a few reasons why mutants might still be live. For instance, if a selfdestruct mutant contains an insertion of a selfdestruct statement, then the contract will be destroyed if it is executed. Now, if this happens with a mutant while testing, then if the contract is used again, it is highly likely that the mutant will be killed. However, if the contract is never called or used again, then there is a chance that the mutant will not be detected and considered live. Developers are typically not going to check for instances where a selfdestruct call may be executed, especially if the selfdestruct call was never in the function or contract to begin with. This is an interesting mutant because it is hard to determine the reasonability of actually testing for this situation. On the other hand, if a selfdestruct call is removed from a function, then the contract will still exist. When these mutants are considered live, we assume that it is live because the test suite never attempts to use the contract again, making an assumption that the contract was destroyed successfully. In any live insert selfdestruct statement (e.g. *selfdestruct(address(0x123)));* the program state will change drastically right after the program has finished executing and should affect the ending program state as well. This indicates that live mutants that cannot detect the selfdestruct mutants do not reach the necessity or sufficiency requirements of testing. This is because the test suite does not correctly check the state of the program immediately after executing the selfdestruct (or when it is removed) and at the end of all execution.

5.2.7 Exception-Handling Mutants

Exception-handling mutants were quite prevalent throughout our experiment. When inserting an exception statement into the contract code, it likely will cause an error to be thrown in an unexpected spot during the program execution. This exception should propagate to the test suite most of the time and should be easily detectable by the test suite in most instances. However, like the event-handling mutation operators, there are many instances where it is unusual for an exception statement to be reached. These instances most often cannot be easily targeted by developers in a test suite and are just rather inserted there in case of extreme circumstances. We believe that this is the reasoning as to why there are so many live mutants of this type. Take the example exception-handling statement: *require(msg.sender == validatorAddress);*. In Deviant, a delete mutation operator would be applied to this statement, causing the condition to never be checked. When this mutant is live, it suggests that the test suite only assumes the happy-path scenario, meaning that it does not meet the necessity requirements, ignoring the state of the program immediately after the mutated statement has executed (or in this case not been executed).

5.2.8 Modifier Mutants

We also received surprising results with our modifier mutants. Typically, it would make sense for preconditions to be checked by tests. As modifiers tend to be focused on checking preconditions before the execution of a function, we would have believed the number of live mutants for this mutant type to be much smaller. However, we believe that developers are either considering only the happy-path scenario or the state that causes the modifier to fail is hard to achieve. For example, if a modifier checks to see if the address of the function callee is correct (e.g. *if(msg.sender == correctAddress)*), then it may be likely that the test suite never attempts to call the function from an invalid address. This could imply that the reachability of the “false” branch is hard to reach in the modifier function, indicating that removing the modifier will not drastically affect the program while it is running in the test suite. However, if the test suite does attempt to use an incorrect address, it would likely cause an exception somewhere in the program, causing a test case to fail. In our case, it does not seem to be highly-likely that this will happen.

5.2.9 Function Type Mutants

Again, our expectations were different from the actual results that we received from these mutant types. In our evaluation, we noticed that the live mutants typically existed when the function types that were originally implemented might have either been designed improperly or are just hard to test. For instance, pure functions are considered to be a subset of view functions. This is because view functions can read the state of the program, but not modify it. However, it is not enforced that a view function read the state of the program. This means that a pure function, which cannot read or modify state, can be modified to a view function without error. This situation is hard to test and does not provide much functional difference. However, we consider the state of the program to be different with this applied mutation, that is because the function can still read from state when the original program did not intend to. Our deleted payable mutants were live primarily if a test suite did not attempt to send ether to that function. This is because a function cannot receive ether if it does not have the payable keyword attached to it. When transfer or send attempts to send ether to a non-payable function it will fail. However, if a contract was using send and did not do proper exception-handling, then it possibly may lead to the mutant being considered live.

5.2.10 Function Visibility Mutants

The results for the function visibility mutants were often mixed. In most cases, if the visibility of the original function versus the mutant were semantically different, it would in most cases cause an exception that would propagate to the test suite. These mutants were almost always marked as killed during the experiment. However, if the visibility was semantically similar (e.g. internal and private), then the mutants would often times be considered live. For example, if a function is modified from public to external and then is only called externally by the test suite, then this mutant will be marked as live. These results primarily indicate one thing about the test suites that were provided: that the test suites do not directly test the visibility of functions. Most of the time when the mutants were killed, it wasn't directly because the function visibility was being tested, it was rather that the function could no longer be called from the correct context. It makes sense as to why this would not often be tested as much as some of the other programming constructs, considering that the effort of testing the visibility would require extra effort and typically isn't related to the typical workflow of the application.

5.2.11 Library Function Visibility Mutants

The results of the library function visibility mutants were practically identical to the function visibility mutants that were discussed above. This makes sense as the semantics of the visibilities between libraries and contracts are hardly different, rather just libraries are slightly more limited. Of course, are numbers show the number of live mutants were much lower, but this is simply because the number of generated mutants for libraries were much smaller. This is because the actual number of libraries in an application tended to be much smaller when compared to the overall number of Solidity smart contracts.

5.2.12 State Variable Visibility Mutants

Again, the results of the state variable visibility mutants were practically identical to the other visibility mutants. There were a high number of live mutants in this case, but as explained earlier, the semantic difference between the visibilities can be hard to test in a test suite. It is uncommon for developers to introduce tests that directly target accessing state variables, just as it is uncommon for developers to introduce tests that target the visibility of functions.

5.3 Threats to Validity

In any software engineering related experiment, there are several threats to validity that must be considered and addressed. This subsection gives an overview of three threats to validity of our experiment, namely: external, internal, and construct. We also explain how we address these threats and explain our decision-making process.

External: An external threat would be that our experiment did not cover real-world Solidity applications. For our evaluation, we experimented on five subject programs. We determined our subject programs based on complexity (lines of code), popularity, and type of application. By considering these attributes, we are able to cover a variety of programs that represent the different types of applications that can be built in Solidity, along with covering the features that are unique to Solidity and Ethereum.

Internal: A potential threat to the validity of our experiment is that our own tool may have faults. We have addressed this threat by using well-known and commonly adopted libraries that are used by the Solidity developer community. We have also taken into consideration the mutants that do not successfully compile. These mutants are considered invalid and ultimately do not count towards our experimental results.

Construct: A threat to the construct of our experiment is that our test does not accurately measure what we are claiming. To address this threat, we have primarily focused on testing the Solidity feature, as that is the primary contribution of our research.

On our subject programs that are smaller in scale, we have used all available mutation operators and have mutated every Solidity file in the project. For our larger scale (aragonOS and OpenZeppelin) projects, we focused only on using the unique Solidity mutation operators. We only used these mutation operators because of the performance overhead to run these test suites and to ensure that we could get enough experimental data for our Solidity mutation operators. For OpenZeppelin, we selected Solidity program files based on the unique Solidity program constructs that existed in the project. Again, we did this because of the performance overhead of running a large project’s test suite repeatedly (in our case 439 times)

CHAPTER SIX: CONCLUSION

We have presented our mutation testing tool, Deviant, for Solidity smart contracts. Our mutation testing tool is designed to target the unique features of Solidity, taking into consideration the design of the Ethereum blockchain. We have also implemented many of the traditional mutation operators that exist in other programming language's mutation testing tools.

We have applied our mutation testing tool to five different open-source Solidity projects with varying complexity. We have determined that in each of these Solidity projects, the test suites that were provided are not adequate enough according to the mutation scores that they received. We have also done preliminary analysis as to why some types of live mutants are more prevalent. In most cases, we believe that these common live mutants typically exist because of the difficulty of testing certain programming constructs or because the developers make assumptions about the state of the contracts that have been used.

For future work, we want to do an in-depth analysis as to what tests can kill certain types of mutants in Solidity. We will be using the generated mutants from this experiment to evaluate the mutants themselves to further understand why these mutants are being missed by the test cases. We can then manually generate test cases that target these live mutants. By doing this, we can determine what test cases can kill certain types of mutants.

REFERENCES

[1] A. Narayanan, J. Bonneau, E. Felten, A. Miller, S. Goldfeder. Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. Princeton: Princeton University Press 2016.

[2] S. Nakamoto. Bitcoin: A peer-to peer electronic cash system (2008). http://bitcoin.org/pdf.

[3] N. Atzei , M. Bartoletti , T. Cimoli, A Survey of Attacks on Ethereum Smart Contracts SoK. Proceedings of the 6th International Conference on Principles of Security and Trust, April 22-29, 2017

[4] Understanding the DAO attack. http://www.coindesk.com/understanding-dao-hack-journalists/

[5] Z. Wan, D. Lo, X. Xia, and L. Cai. 2017. “Bug characteristics in blockchain systems: a large-scale empirical study”. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 413–424.

[6] “muJava” [Online]. Available: https://cs.gmu.edu/~offutt/mujava/

[7] “Stryker” [Online]. Available: https://stryker-mutator.io/

[8] Y. Ma, J. Offutt, Y. Kwon. “MuJava : An Automated Class Mutation System”. *Journal of Software Testing, Verification and Reliability*, 15(2):97-133, June 2005.

[9] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, “The Class-Level Mutants of MuJava,” in *Proceedings of the 2006 international workshop on Automation of software test - AST ’06*, 2006, p. 78.

[10] Y. Ma, Y. Kwon and J. Offutt. “Inter-Class Mutation Operators for Java”. *Proceedings of the 13th International Symposium on Software Reliability*

Engineering, IEEE Computer Society Press, Annapolis MD, November 2002, pp. 352-363.

[11] R. DeMillo, R. Lipton, F. Sayward. “Hints on test data selection: Help for the practicing programmer”. *IEEE Computer*, 11(4):34-41. April 1978.

[12] V. Buterin, “A Next Generation Smart Contract & De-centralized Application Platform”, Ethereum White Paper https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf/, 2013.

[13] “Solparse” [Online]. Available: https://github.com/duaraghav8/solparse

[14] “Solmeister” [Online]. Available: https://github.com/duaraghav8/solmeister

[15] “Truffle” [Online]. Available: https://github.com/trufflesuite/truffle

[16] A. Mavridou, A. Laszka, “Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach”, *22nd International Conference on Financial Cryptography and Data Security (FC 2018).*

[17] I. Grishchenko, M. Maffei, and C. Schneidewind, “A Semantic Framework for the Security Analysis of Ethereum smart contracts”, *International Conference on Principles of Security and Trust*

[18] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, B. Roscoe, “ReGuard: finding reentrancy bugs in smart contracts”, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, May 27-June 03, 2018, Gothenburg, Sweden

[19] R. Martin. “Java and C++: a critical comparison”, *In Java Gems*, Dwight Deugo (Ed.). Cambridge Sigs Reference Library Series, Vol. 9. Cambridge University Press, New York, NY, USA 51-68.

[20] K. Barrett, B. Cassels, P. Haahr, D. Moon, K. Playford, “A Monotonic Superclass Linearization for Dylan”. *OOPSLA '96 Conference Proceedings*. ACM Press. 1996-06-28. pp. 69–82.

[21] “Alice” [Online]. Available: https://github.com/alice-si/contracts

[22] “MetaCoin” [Online]. Available: https://github.com/truffle-box/metacoin-box

[23] “MultiSigWallet” [Online]. Available: https://github.com/gnosis/MultiSigWallet

[24] "aragonOS” [Online]. Available: https://github.com/aragon/aragonOS

[25] “OpenZeppelin” [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-solidity

[26] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," in *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 371-379, July 1982.

[27] D. Schuler, V. Dallmeier, and A. Zeller, “Efficient mutation testing by checking invariant violations”, *In Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09)*, ACM, New York, NY, USA, 69-80.

[28] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. “Dynamically discovering likely program invariants to support program evolution”. *IEEE Transactions on Software Engineering,* 27(2):99–123,Feb. 2001.

[29] J. Offutt, G. Rothermel, C. Zapf, “An experimental evaluation of selective mutation”. *In Proceedings of the 15th international conference on Software Engineering (ICSE '93),* IEEE Computer Society Press, Los Alamitos, CA, USA, 100-107.

[30] A. Mathur, “Performance, effectiveness, and reliability issues in software testing”, *COMPSAC (1991).*

[31] J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. “An experimental determination of sufficient mutant operators”, *ACM Trans. Softw. Eng.Methodol*. 5, 2 (April 1996)

[32] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, S. Zanella-B´eguelin. “Formal verification of smart contracts: Short paper”, In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS ’16,* pages 91–96, New York, NY, USA, 2016. ACM. 4

[33] W. Chen , Z. Zheng , J. Cui , E. Ngai , P. Zheng , Y. Zhou, “Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology”, Proceedings of the 2018 World Wide Web Conference, April 23-27, 2018, Lyon, France

[34] L. Luu, D. Chu, H. Olickel, P. Saxena, A. Hobor, “Making smart contracts smarter”. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pages 254–269. ACM, 2016. 4

[35] A. Mavridou, A. Laszka, “Tool demonstration: Fsolidm for designing secure ethereum smart contracts”, CoRR, abs/1802.09949, 2018. 4

[36] N. Popper, “Hacker may have taken $50 million from cybercurrency project”, The New York Times, https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html,

[37] Y. Wang, Q. Malluhi, “The Limit of Blockchains: Infeasibility of a Smart Obama-Trump Contract”, https://eprint.iacr.org/2018/252.pdf

[38] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, G. Rosu, “Kevm: A complete semantics of the ethereum virtual machine” [Online], Available: http://hdl.handle.net/2142/97207

[39] “FSolidM” [Online]. Available: https://github.com/anmavrid/smart-contracts

[40] “F*” [Online]. Available: https://fstar-lang.org

[41] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, and G. Fraser. "Are Mutants a Valid Substitute for Real faults in Software Testing?" *In Proc. of the Symposium on the Foundations of Software Engineering (FSE’14)*, 654-665, Hong Kong, November 2014.

[42] “coveralls” [Online]. Available: https://coveralls.io/

[43] “solidity-coverage” [Online]. Available: https://github.com/sc-forks/solidity-coverage

APPENDIX A

A.1 General Intra-Statement Level Mutation Operators

Beyond the Solidity-specific mutation operators that were discussed, we have implemented several other traditional mutation operators. These mutation operators were inspired or based off of mutation operators from Styker[7] and muJava[6].

Binary Expression Operators: Binary expressions consist of two operands and one operator. For mutation purposes, we only consider the operator itself in binary expressions as the operands themselves tend to be literal or variables that will be handled by other mutation operators. These mutation operators are quite simple, but do in fact represent many common programming mistakes as they often represent minor logical or arithmetic errors. Also, these mutation operators tend to generate many mutants when compared to some of the other mutation operators. That is because for each instance of a binary expression, there are usually several valid mutation operators that can applied to an individual. It is also because binary expressions tend to be quite plentiful in programming. Table A.1 lists the binary mutation operators that are implemented in Deviant.

Table A.1 Binary Mutation Operators

Mutation Operators	Abbreviations	Fault Types
+ → [-, *, /, %]	PTMN, PTMU, PTD, PTMD	Arithmetic Error
- → [+, *, /, %]	MNTP, MNTMU, MNTD, MNTMO	Arithmetic Error
* → [+, -, /, %]	MUTP, MUTMN, MUTD, MUTMO	Arithmetic Error
/ → [+, -, *, %]	DTP, DTMN, DTMU, DTMO	Arithmetic Error
% → [+, -, *, /]	MOTP, MOTMN, MOTMU, MOTD	Arithmetic Error
< → [>, <= , >=, ==, !=]	LTGT, LTLTE, LTGTE, LTE, LTNE	Conditional Error
> → [<, <= , >=, ==, !=]	GTLT, GTLTE, GTGTE, GTE, GTNE	Conditional Error
<= → [>, < , >=, ==, !=]	LTEGT, LTELT, LTEGTE, LTEE, LTENE	Conditional Error
>= → [>, <= , <, ==, !=]	GTEGT, GTELTE, GTELT, GTEE, GTENE	Conditional Error
== → [>, <= , >=, <, !=]	EGT, ELTE, EGTE, ELT, ENE	Conditional Error
!= →[>, <= , >=, ==, <]	NEGT, NELTE, NEGTE, NEE, NELT	Conditional Error
&& → ||	AOR	Conditional Error
|| → &&	ORA	Conditional Error
& → [|, ^]	BABOR, BAXOR	Binary Arithmetic Error
| → [&, ^]	BORBA, BORXOR	Binary Arithmetic Error
^ → [&, |]	XORBA, XORBOR	Binary Arithmetic Error
<< → >>	LSRS	Binary Arithmetic Error
>> → <<	RSLS	Binary Arithmetic Error

Unary Expression Operators: Like binary expressions, unary expressions involve operands and operators. However, a unary expression involves only one operator and one operand. These operators, like binary expression operators, also tend to generate many mutants when compared to other mutation operators. These operators are especially important when considering the state of a contract. For example, Solidity has been a popular source of implementing decentralized online voting systems. Many of which rely on the unary update or decrement operators that exist. Table A.2 lists the unary mutation operators.

Table A.2 Unary Mutation Operators

Mutation Operators	Abbreviations	Fault Types
- → DELETE	DLMN	Arithmetic Error
~ → DELETE	DLBN	Binary Arithmetic Error
! → DELETE	DLN	Conditional Error
++ → [DELETE, --]	DLINC, INCDEC	Arithmetic Error
- - → [DELETE, ++]	DLDEC, DECINC	Arithmetic Error

Assignment Expression Operators: Assignment expressions consist of an assignment of a variable to some value, either another variable or some literal. Beyond the ordinary assignment expression, Solidity as well supports compound assignments. Compound assignments are an assignment along with an arithmetic operator. When performing a mutation with an assignment expression operator, it is important to consider the type of variable that is in the assignment expression. For instance, many of the compound string assignments that exist in languages such as Java, are not supported in Solidity. Instead, these mutation operators are only relevant for numerical types such as integers. However, numerical types tend to be the dominant literal that exists in Solidity, meaning that there will be plenty of instances where assignment expression operators will generate mutants. Table A.3 lists the assignment mutation operators.

Table A.3 Assignment Mutation Operators

Mutation Operators	Abbreviations	Fault Types
*= → [=, /=, +=, -=, %=, |=, &=, ^=]	EMUE, MUEDE, MUEPE, MUEMNE, MUEMOE, MUEOE, MUEAE, MUEXOE	Arithmetic Error Ether Lost in Transfer
/= → [=, *=, +=, -=, %=, |=, &=, ^=]	EDE, DEMUE, DEPE, DEMNE, DEMOE, DEOE, DEAE, DEXOE	Arithmetic Error Ether Lost in Transfer
+= → [=, /=, *=, -=, %=, |=, &=, ^=]	PEE, PEDE, PEMUE, PEMNE, PEMOE, PEOE, PEAD, PEXOE	Arithmetic Error Ether Lost in Transfer
-= → [=, /=, +=, *=, %=, |=, &=, ^=]	MNEE, MNEDE, MNEPE, MNEMUE, MNOE, MNEOE, MNEAE, MNEXOE	Arithmetic Error Ether Lost in Transfer
%= → [=, /=, +=, -=, *=, |=, &=, ^=]	MOEE, MOEDE, MOEPE, MOEMNE, MOEMUE, MOEOE, MOEAE, MOEXOE	Arithmetic Error Ether Lost in Transfer
|= → [=, /=, +=, -=, %=, *=, &=, ^=]	OEE, OEDE, OEPE, OEMNE, OEMOE, OEMUE, OEAE, OEXOE	Arithmetic Error Ether Lost in Transfer
&= → [=, /=, +=, -=, %=, |=, *=, ^=]	AEE, AEDE, AEPE, AEMNE, AEMOE, AEOE, AEAE, AEXOE	Arithmetic Error Ether Lost in Transfer
^= → [=, /=, +=, -=, %=, |=, &=, *=]	XOEE, XOEDE, XOEPE, XOEMNE, XOEMOE, XOEOE, XOEAE, XOEMUE	Arithmetic Error Ether Lost in Transfer
= → [*=, /=, +=, -=, %=, |=, &=, ^=]	EMUE, EDE, EPE, EMNE, EMOE, EOE, EAE, EXOE	Arithmetic Error Ether Lost in Transfer

Literal Expression Operators: Literals in code are a fixed value associated with a type, such as an integer or string. In terms of mutation, literal operators modify the fixed value to some other value. In Deviant, the mutation operators that associate with numerical types typically either mutate to a zero or random non-zero value. The idea behind mutating to one of these two values is that programs often times have specific behavior defined for zero values and non-zero values. Strings are handled differently, in that their content is either deleted or just modified. Arrays are handled by either changing the length, making them dynamic, or making them fixed. In some instances, this might not have much effect on the behavior of the application, but by changing these attributes of the array there is a chance that the behavior is affected in some manner. Boolean operators are quite simple as they are simply just change true to false and false to true.

Table A.4 Literal Mutation Operators

Mutation Operators	Abbreviations	Fault Types
String → [Delete, Add Content]	DLST, ADDST	Incorrect Literal
Integer → [Non-Zero, Zero]	INTNZ, INTZ	Arithmetic Error Incorrect Literal
Hexadecimal → [Zero, Non-Zero]	HXZ, HXNZ	Arithmetic Error Incorrect Literal
Bytes Array → [Modify Length]	BTAML	Overflow Arithmetic Error
Fixed Array → Dynamic Array	FADA	Memory Storage Change
Dynamic Array → Fixed Array	DAFA	Overflow
True → False	TF	Conditional Error
False → True	FT	Conditional Error

A.2 General Intra-Function Level Mutation Operators

Beyond the several Solidity-specific mutation operators, there are four more general mutation operators that are applicable to the intra-function level. These mutation operators include: Delete Statement (DS) or Delete Block (DB). In most cases, these mutation operators work similarly to the mutation operators that were defined above. However, these mutation operators are more general and do not specifically target unique features of Solidity. In fact, the Solidity-specific mutation operators are rather a subset of these operators.

A.3 General Intra-Module Level Mutation Operators

Intra-Module Variable Modification Operators: These operators deal with the state variables that exist in a parent-child relationship between contracts. In this category, these variables are those that the child inherits from the parent. The specifics of the mutation operators are as follows:

- *Hiding Variable Insertion (HVI):* This mutation operator applies when a parent contract contains a state variable that isn't declared directly within the child contract. This mutation operator inserts a variable of the same name into the child contract. By performing this mutation, when that variable is ever used in the context of the child contract, the child's version of the variable will be used. This can of course cause a variety of potential errors to happen.
- *Hiding Variable Deletion (HVD):* This mutation operator is applied when a parent and child contract contain a global variable with the same name. In this case, the mutation operator deletes this global variable from the child contract. When this mutation operator is applied, it can potentially cause the child contract to use the parent contract version of the variable. Of course, this can cause a variety of potential errors to happen.

General Intra-Module Level Mutation Operators: Beyond the mutation operators listed previously, there are also operators that deal with the relationship between a parent contract and child contract beyond the scope of hiding variables. These mutation operators are based on the class-level mutation operators provided by muJava, but with the primary focus being on the behavior of Solidity instead. These operators are considered intra-module level because they mutate the content within the actual contract itself (e.g., functions that are considered overriding). The overriding function operators mutate the relationship between a child and parent contract in relation to the functions that share the same name. For example, the Overriding Function Deletion (OFD) operator deletes the signature and body of an overriding function that exists in a child contract. By doing this, the parent contract's version of the function will instead be used when that function is called from the context of the child contract. This will cause unexpected behavior as a developer would expect the child's overriding function to be used instead. The super keyword mutation operators are again involved in the relationship that exists between a parent and child in relation to the programming constructs that have the same shared name between the two contracts. Now, what is different between these super keyword operators and the other mutation operators is that they do not delete or remove the child or parent's version of the programming construct. Instead, these mutation operators take an individual instance of a programming construct with a shared name and either insert or delete the super keyword. What this mutation does is cause the parent or child's version of the programming construct to be used whenever the original intention was to use the opposite. The type-based operators are mutation operators that modify instances where a parent or child can replace an instance of a contract that is being cast. For example, the Cast Type Change (CTC) operator takes an instance of an address or contract being cast to another contract and takes a valid (typically a parent or child) contract and replaces the cast with the other contract instead. This will cause the cast contract to behave as the newly casted type instead. Table A.5 lists the general intra-module level mutation operators.

Table A.5 Intra-Module Level Mutation Operators

Mutation Operators	Abbreviations	Fault Types
Overriding Function Deletion	OFD	Incorrect Function Behavior
Overriding Function Calling Position Change	OFCP	Incorrect Function Behavior
Overriding Function Rename	OFR	Incorrect Function Behavior
Super Keyword Insertion	SKI	Incorrect Function Behavior Incorrect Variable Behavior
Super Keyword Deletion	SKD	Incorrect Function Behavior Incorrect Variable Behavior
Type Cast Operator Insertion	TCOI	Type Cast Error Incorrect Function Behavior Incorrect Variable Behavior
Type Cast Operator Deletion	TCOD	Type Cast Error Incorrect Function Behavior Incorrect Variable Behavior
Cast Type Change	CTC	Type Cast Error Incorrect Function Behavior Incorrect Variable Behavior