THE UNIVERSITY
of EDINBURGH

Automated Testing for Solidity Smart Contracts

Sefa Akca

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2022

Abstract

Blockchains are the underlying technology for making secure online transactions using cryptocurrencies such as Bitcoins and Ethers. Executing, verifying, and enforcing credible transactions on permissionless blockchains is done using smart contracts. Smart contracts are publicly available programs that run on the Ethereum network and can be implemented by Solidity language. Solidity is the most popular programming language to implement smart contracts. Smart contracts help exchange money, property, shares or anything of value in a transparent, conflict-free way while reducing transaction costs associated with third-party contractors.

A major challenge in developing smart contracts is to guarantee that they are correct and free of security weaknesses, since bugs in their implementation may result in significant financial losses. For example, a single line that triggered integer overflow vulnerability in the BeautyChain (BECToken) contract caused a large number of stolen tokens - worth approximately $12 million. In the last few years, analysis and testing of smart contracts have raised considerable interest, and numerous techniques have been proposed to check the presence of vulnerabilities in them. However, the security and correctness of smart contracts is still a big question in academia and industry because of the problems in proposed techniques such as high false-positive rate, path explosion, and lack of support for Solidity constructs.

This thesis presents novel approaches for automated test input generation and test effectiveness measurement for Solidity smart contracts. To achieve this, we address significant challenges related to smart contract execution, namely Solidity language-specific data and features, and the extent of vulnerability detection. The thesis makes the following four contributions within automated testing.

First, we propose an analysis, instrumentation, optimisation and code generation framework, SIF, using pre-defined helper functions that do not require any expert Solidity programming. This framework is capable of providing an interface to easily and effectively understand, manipulate and analyse Solidity code.

Second, we present an automated framework, SolAnalyser, for detecting vulnerabilities over Solidity smart contracts. Our framework automatically inserts relevant assert statements as pre and post conditions for each arithmetic operation in the contract. Then, it generates a smart contract with relevant property assertions as output. Finally, our framework executes the generated smart contract with property assertions with automatically generated test inputs (from Contribution 3), and reports vulnerabilities, if any.

Third, we implement different test input generation techniques for Solidity smart contracts. The techniques we implemented are based on fuzzing, genetic algorithm (GA), and satisfiability modulo theories (SMT) solver. Fuzzing relies on random input generation depending on contract interface. GA imitates the natural evolution to produce test input. SMT solver aims to solve given constraints for a practical subset of inputs.

As the fourth contribution, we implement a framework, CovCal, to evaluate the effectiveness of different test input generation techniques. CovCal analyses them with respect to different coverage metrics, which are specific to Solidity structure, and fault-finding ability.

We evaluate our frameworks, i.e., SIF, SolAnalyser and CovCal, and the testing techniques under different smart contract datasets that contain 1755 smart contracts and a wide variety of the Solidity constructs. We can infer the following from these experiments: (1) SIF is capable of supporting all the Solidity constructs, correctly instrumenting the Solidity code, and is user-friendly. (2) SolAnalyser is capable of detecting defined vulnerability types. (3) CovCal is capable of measuring the effectiveness of the test inputs automatically, in terms of code coverage and fault-finding capability in the Solidity smart contracts. (4) All of these techniques are able to generate valid test inputs for the Solidity smart contracts and detect any vulnerabilities in the tested contracts. All of these implementations and evaluations are open-sourced and have been used by other research groups.

Lay Summary

A blockchain is a decentralized database that is shared across computer network nodes. It holds information in digital format. Blockchains are best known to be an underlying technology for making secure online transactions using cryptocurrencies such as Bitcoins and Ethers. Smart contracts are used to execute, validate, and enforce credible transactions on permissionless blockchains. Smart contracts are small programs stored on the Etherum network and implemented by different high-level programming languages such as Solidity. They run when predefined conditions are met. Contracts can send, hold, receive an amount of virtual coin that motivates attackers. This has made checking their security and trustworthiness critical.

In this thesis, we propose novel approaches to generate automated test inputs and measure the effectiveness of the test inputs by taking into account the characteristics of the smart contracts, programming language and execution environment. Our techniques are useful to generate test inputs with high code coverage and fault finding capabilities and to detect and report vulnerabilities, if any, in contracts to developers and testers. In summary, our approaches help provide confidence in the security and trustworthiness of Solidity smart contracts.

Acknowledgements

First, I would like to thank my supervisor, Dr. Ajitha Rajan, for her invaluable guidance and support. I am very grateful for her guidance and encouragement throughout this challenging and long journey. She was much more than a supervisor for me.

I would also like to thank the Republic of Turkey, Ministry of National Education for their full support which makes my dream of studying abroad possible.

I would also like to thank Mustafa Kemal ATATURK who gave inspiration not only for me but also for the nation. I have been followed and will follow the motto he teach us “The supreme guide in life is science”.

I would also like to thank my colleagues: Chao Peng, Vanya Yaneva, Foivos Tsimpourlas and Nick Louloudakis. I had great fun working together. Thank you to my friends Tayfun Aktas, Mehmet Erdem Caglar, Chao Peng, Aryan Kaushik, Burak Sahin, Abdulhamit Tayfur, Beyza Tayfur, Engin Sonmez, Laura Schneider, Pauliina Vuorinen, Ahmet Narman and Merve Orme for making my long journey easier and funnier.

I would also like to thank my mum, dad, sisters and their husbands for their emotional support and eternal belief in me.

Last but not least, my dear beloved wife Asena Avci Akca who is my rock. I am deeply grateful for having you in my life. Without you, it would be impossible for me to finish this journey.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Peng, C., Akca, S. and Rajan, A., 2019, December. SIF: A framework for solidity contract instrumentation and analysis. In 2019 *26th Asia-Pacific Software Engineering Conference (APSEC) (pp. 466-473). IEEE.*
- Akca, S., Rajan, A. and Peng, C., 2019, December. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In 2019 *26th Asia-Pacific Software Engineering Conference (APSEC) (pp. 482-489). IEEE.*
- Akca, S., Peng, C. and Rajan, A., 2021, October. Testing Smart Contracts: Which Technique Performs Best?.*In Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (pp. 1-11).*

(*Sefa Akca*)

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A blockchain is a distributed ledger that contains unmodifiable records called blocks that can be linked to previous blocks. Blockchains are the underlying technology for making secure online transactions using cryptocurrencies such as Bitcoins and Ethers. Smart contracts enable the execution, verification, and enforcement of credible transactions on blockchains [107], which are written by the buyer and seller [19].

Smart contracts help change money, property, shares, or any kind of value in a transparent, conflict-free way while reducing transaction costs associated with third-party contractors. Solidity is a popular object-oriented and high-level language for writing smart contracts [29, 111] and can be compiled to bytecode for execution on the ethereum network. With the increased use of smart contracts across application domains, there is a crucial need for a unified framework that supports and facilitates Solidity code analysis, understanding, transformation, and development of tools for verification and testing that provide strong security guarantees.

A key challenge in developing contracts is to ensure that they are correct and free of security vulnerabilities, as bugs in their implementation may result in substantial financial losses. However, their security and trustworthiness are still in question. Many vulnerabilities in contracts have been reported [10, 22]. For instance, one of the most known attacked is decentralized autonomous organization (DAO) [100], due to unsafe design choices resulted in losses of approximately $50 million. However, DAO is not the only attack that resulted in financial losses. For instance, due to bugs in the contracts, users lost over $145 million in the six years (2011- 2017). Notably, more than these losses happened in the following two years (2018, 2019) [4].

1.1 Smart Contract Bugs

After smart contracts have been introduced, the popularity of them has increased sharply. Contracts store a large number of virtual coins, which motives attackers.

Contracts can contain a variety of bugs. Some of them such as integer overflow/underflow are similar to bugs in traditional programming languages. On the other hand, some of the bugs are specific to contracts and these bugs are explained in the Background section in detail. Due to financial losses related to bugs, the testing and analysis process of the contracts caught the attention of developers and researchers. Therefore, many researchers in academia and industry have been developed tools and frameworks to analyse and detect the vulnerabilities in the contracts.

1.1.1 Smart Contract Testing

Until the last couple of years, using static analysis techniques in the smart contract domain was more common than testing techniques. Static analysis techniques, although effective in detecting security vulnerabilities in traditional programming languages, are facing some crucial drawbacks in Smart contract domains such as high false-positive rate, lack of support for Solidity construct, and time complexity [32, 44, 97].

Testing techniques in the smart contract domain have started to emerge for the last couple of years, and our contribution in developing smart contract testing tools is among the first in the field. One of the main concerns in the testing process, the effectiveness and quality of the testing techniques depends on the quality of the generated input.Testing techniques that have proven to be effective in other domains can be used in the smart contract domain to tackle this concern. In addition, a metric such as branch coverage or an additional tool such as a constraint solver can be merged with a testing technique to increase the quality of the input. Currently, however, only a limited number of testing techniques have been used in the smart contract field, and the effectiveness of the testing techniques has still not been examined. In addition, the drawbacks faced by static analysis techniques in the smart contract domain may be solved by testing techniques. The high-false positive rate is not a concern for the testing process since generated test input set will be executed during the run-time with the software under test. The time-complexity problem in static analysis techniques might be solved with optimized test generation techniques. Full support for Solidity construct in testing techniques is not trivial. Still, it is not a big threat as static techniques have, since the solidity specific data types such as balance and account can be adapted to the traditional data types

during test generation. If data adaptation is not possible, we can feed to software/system under test (SUT) during the execution with the generated inputs during the execution.

1.2 Problem Statement

Unlike traditional distributed application platforms, contract platforms such as Ethereum [116] operate in open networks that allow arbitrary participants to join. Thus, their execution is vulnerable to attempted manipulation by arbitrary adversaries, as opposed to traditional permissioned networks where the threat is more restricted. A key challenge in developing contracts is to ensure that they are correct and free of security vulnerabilities, as bugs in their implementation may result in substantial financial losses.

Automated testing techniques are commonly used to detect vulnerabilities in traditional programming languages such as C, C++, Java, etc. and we believe they would be well suited to detecting vulnerabilities in smart contracts. Solidity is a newly defined high-level language. The syntax of the Solidity is similar to JavaScript language. However, they have many essential differences. (1) JavaScript is an interpreted language whereas Solidity is a compiled language. (2) Solidity is a statically typed language whereas java-Script is a dynamically typed language. (3) In contrast to JavaScript, if a developer uses complex types such as arrays, structs, etc., the typing system requires the developer to define the memory location of the data that will be live in Solidity. (4) Solidity can interact with any other smart contracts deployed on the Ethereum network without permission. However, it is not a case for JavaScript language-based code. JavaScript code can interact with another code, if it holds an application programming interface (api) and permission to access interacted code. (5) Solidity also contains language-specific data and function types. Plus, the running environment of Solidity is different from traditional languages in some aspects. (6) Last but not least, some of the vulnerability types in Solidity are not the same as the traditional programming languages. Although there are many differences between the Solidity and traditional languages, Solidity language and vulnerabilities may be translated to traditional languages. That is, instead of implementing test case generation tools for Solidity smart contracts, Solidity language, and vulnerabilities can be translated to traditional languages. However, translation of the Solidity language to another high-level language is not trivial and does not need less effort than implementing new test case generation tools. More importantly, with the increased use of smart contracts across application domains, there is a crucial need for a framework that verifies and tests Solidity code.

In addition, Solidity language is still developing with new features, and it is hard to estimate that every new feature and vulnerability will be fit for translation. Therefore, the author of this thesis decided to propose a framework for smart contract developers to analyse and instrument smart contracts with a unified framework. In addition to that, input generation techniques that are commonly used for traditional languages were proposed for Solidity smart contracts. Lastly, novel and Solidity-specific coverage metrics were presented to evaluate the effectiveness of the generated test inputs.

Adapting the automated test generation process for smart contracts consists of three main subprocesses: Test case generation, test execution, and test effectiveness measurement.

1.2.1 Test input generation

As mentioned in Section 1.1.1 , the variety of the test input generation techniques in the smart contract domain have been limited. Researchers have developed some test input generation tools [48, 56, 81, 118]. However, the effectiveness of these techniques has not been thoroughly examined. In addition, many input generation techniques which are known to be effective in other domains have not been applied in this domain. As a result, the effectiveness of the testing techniques and the applicability of the well-known techniques for smart contracts was one of our research contributions.

1.2.2 Test Execution and Effectiveness

Test effectiveness measurement is used to show the effectiveness of the testing method with respect to a desired criterion [57]. The effectiveness of testing has been traditionally measured in terms of code coverage achieved and fault finding ability [43, 89]. The code coverage measures how much of the code has been exercised by test cases, and the fault finding ability measures how many of the inserted faults are detected by test cases.

There is limited existing work in measuring Solidity code coverage and fault finding [17, 41, 98]. Thus, we decided to define and measure more coverage metrics which are novel and Solidity specific. In addition, we have implemented a new environment to measure the effectiveness of the test cases in the run time.

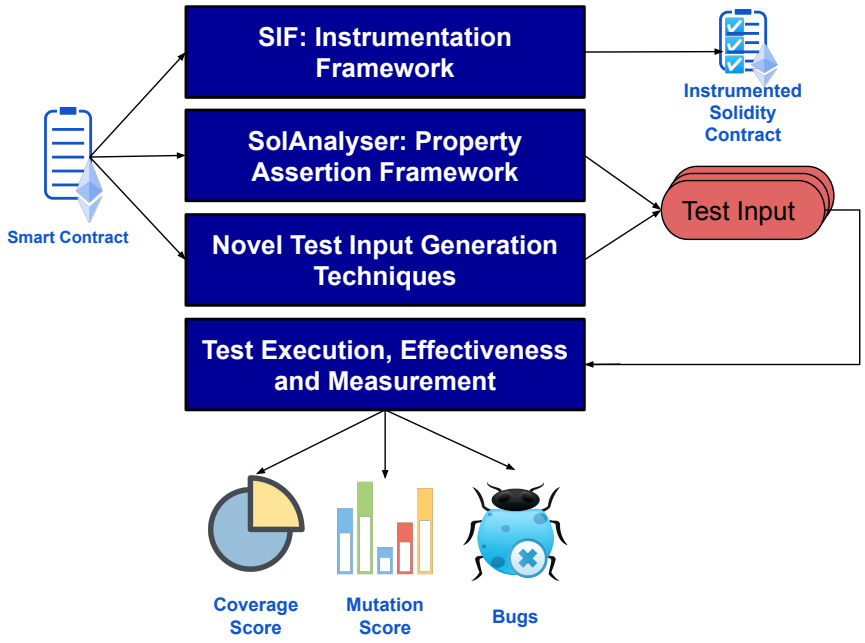

Figure 1.1. Thesis contributions

1.3 Contributions

This thesis makes four main contributions. Figure 1.1 presents these contributions to test Solidity Smart contracts.

1.3.1 Instrumentation and Analysis Framework for Solidity Smart contracts

A Solidity smart contract instrumentation and analysis framework, SIF, is implemented to support for Solidity smart contract developers and testers to build source level techniques for analysis, understanding, diagnostics, optimizations, and code generation. Our motivation to build SIF framework is to understand the Solidity smart contract code before analysing it as LLVM/Clang [64] is doing for C/C++. In addition to that, we also planned to use the SIF framework to modify the smart contracts and to evaluate the test techniques with respect to the mutation killed score. SIF provides an interface for developers and testers to query Abstract Syntax Tree (AST) of Solidity code. This framework is able to automatically instrument or transform Solidity code using pre-defined helper functions and supports all Solidity data and function types. One of the main challenge faced during SIF design and implementation was to generalize the SIF framework for different versions of the compiler. To achieve this, we used C++ classes as our intermediate representation. Different node types and all the associated

information are defined inside the C++ classes. Another challenge faced was to make the SIF framework extendable. We are aware that Solidity is a developing language. Developers and/or testers may need to add extra tools on top of the SIF framework. To achieve this, we implemented a visitor, a before, and an after function. Users only need to update these functions with respect to their needs without changing the core of the framework. SIF is the first instrumentation framework for Solidity smart contracts, it is open-source with many users. This contribution is explained in detail in Chapter 4.

1.3.2 Property Assertion Framework for Solidity Smart contracts

SolAnalyser is a fully automated framework for vulnerability detection over Solidity smart contracts that uses both static and dynamic analysis. This framework is developed for smart contract developers and testers to test their smart contracts before deploying them on the Ethereum network. This framework inserts relevant assert statements as pre and post conditions where necessary then generate a smart contract with property assertions. Lastly, the SolAnalyser framework uses blackbox fuzzing technique to generate test cases for the smart contract with property assertions. The main challenge we faced during the implementation of the SolAnalyser framework was to insert related assertions before and/or after arithmetic operations. To solve this, we used two types of AST files (plain and JavaScript Object Notation format) generated by the Solidity compiler. We traversed the ASTs and checked specific node types and their operand and operators. This contribution is explained in detail in Chapter 5.

1.3.3 Novel Test Input Generation Techniques

We implemented three different test case generation techniques for Solidity smart contracts. The first one is Blackbox fuzzing which is implemented as a part of the SolAnalyser framework. The other two techniques Coverage-Guided Fuzzing with SMT solver (GF), and Genetic Algorithm (GA). These techniques are not new, however, have not been implemented for smart contracts. Our motivations to implement these test input generation techniques to Solidity smart contracts are: (1) BF is the simplest technique to generate random test data according to a distribution for the different inputs. As mentioned in Lay Summary, smart contract codes are fairly simple and not very complicated. Therefore, BF is a good initial point to discover the potential problems in testing techniques for smart contracts. (2) After we observed the limitations of the BF technique, we decided to use the power of the SMT solver by combining BF. Thus,

we decided to implement GF technique. GF is one of the most common techniques and has been used in different domains [8, 54, 60, 90], our implementation is the first in the smart contract domain. (3) GA techniques have been performed in many different domains [6, 9, 13, 30, 38] and the results of the technique are promising.Apart from that, we wanted to use the power of the objectives, which GA has, to achieve higher coverage with limited resources. Thus, we implemented the technique for Solidity language. To our knowledge, there is no existing tool implementing GA for testing smart contracts, our implementation is the first in the smart contract domain. (4) Empirical evaluation of the fuzzing-based and GA techniques is worth to know for our future work in the field. All of the testing techniques we implemented, support all data and function types of Solidity. The key challenge we faced during the implementation of the testing techniques is to adapt the Solidity-specific data and function types for the language we used to implement testing techniques. To achieve this, we presented Java classes for each Solidity-specific data type. The data types which were not adopted as Java classes were fed during the execution. This contribution is explained in detail in Chapter 6.

1.3.4 Test Effectiveness Measurement

To assess the effectiveness of the test case generation techniques, we implemented a fault seeder, an execution environment (ExEn), and a coverage environment (CovCal). Fault seeder tool allows to developer to artificially seed bugs in Solidity smart contracts. Our execution environment, implemented as an extension to Ethereum Virtual Machine (EVM) to analyze the execution traces and report triggered vulnerabilities, helps to determine the fault finding score for the contract under test. Our Coverage Environment is implemented in nodejs as an extension to EVM to analyze the execution trace produced by EVM. CovCal is able to determine four different coverage metrics - Opcode coverage, Branch coverage, Event coverage, and Call coverage. This contribution is explained in detail in Chapter 6.

1.4 Publications

The ideas and results presented in this thesis are based on three previous publications.

1. *Peng, C., Akca, S. & Rajan, A. SIF: A Framework for Solidity Contract Instrumentation and Analysis. In APSEC 2019.* [88]

The framework, SIF, for Solidity smart contract instrumentation and analysis, presented in Chapter 4 was first published in the APSEC 2019.

2. *Akca, S., Rajan, A. & Peng, C. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In APSEC 2019.* [3]

The framework, SolAnalyser, for analysing and testing Solidity smart contract, presented in Chapter 5 was published as research paper in APSEC 2019.

3. *Akca, S., Peng, C. & Rajan, A. Testing Smart Contracts: Which Technique Performs Best?. In ESEM 2021.* [2]

The empirical evaluation of different test generation techniques for Solidity smart contract, presented in Chapter 6 was previously published as research paper in ESEM 2021.

Chapter 2

Background

This chapter explains relevant background information to understand the problem and solutions discussed in this thesis. Section 2.1 with the following subsections explain concepts of Blockchain, Etherum, and Etherum virtual machine, respectively. Section 2.2 explains the structure of the smart contract, language features of Solidity, and well-known vulnerabilities in the Smart contracts with examples. The last section 2.3 describes software testing workflow and concepts involved in testing.

2.1 Blockchain

Blockchains are an underlying technology for securing online transactions using cryptocurrencies like Bitcoin and Ether. Blockchain is distributed ledgers. In other words, blockchain is a public database that can be updated and shared across many computers in a network [106]. *Block* refers to a group of data and state that are sequentially stored. *Chain* refers that each block is linked to its parents.

2.1.1 Ethereum

Ethereum is an open-source, permissionless, and decentralized blockchain network with smart contract functionality [29, 117]. It has the native cryptocurrency Ether (ETH) generated by Ethereum protocol. Ethereum uses a proof of work consensus mechanism. That is, if any user wants to add new blocks to the chain, the user needs to solve a puzzle. To solve this puzzle requires significant computing power, and solving it shows to computational resources of the user. This process is known as mining.

In the Ethereum system, every user holds a single and standard computer called

Ethereum Virtual Machine (EVM). All user on the Ethereum network agrees to keep a copy of the state of EVM [52].

2.1.1.1 Ethereum Virtual Machine

Ethereum Virtual Machine (EVM) is a software platform based on blockchain. Developers can create decentralized applications (Dapps) using EVM [52]. EVM executes the tasks using opcodes, which is similar to assembly language. Every opcode has a special meaning and is encoded to bytecode, and it divides into its bytes when it executes a task.

Accounts in Ethereum can send Ether or binary data via a transaction from one account to another. Etherum has two types of accounts. These are external and contract. EVM treats both of them equally. The external account is controlled by Public-private key pairs. The contract account is controlled by code kept within the account.

As mentioned before, Ethereum is a decentralized network. It means there is no central authority on the network. All contracts in the network are executed on Ethereum nodes. This system could cause a risk of a malignant user slowing down the network intentionally. A gas system is used on the network to protect the network from malignant user attacks. On the EVM, each transaction is charged a certain amount of gas based on its operation. The gas system aims to eliminate unnecessary operations in transactions. The gas fee schedule for every opcode is published in the Ethereum yellow paper [117].

EVM has three separate memory areas. The first area is called as “storage area” that is a persistent memory area located inside the account. This area is created during the process of creation of a contract. Except for own storage, any contract can neither read nor write to any other contract’s storage. The second area is the “memory”. That is linear and contains temporary variables. Memory can be addressed byte-level, but reading and writing operations in the memory are limited - reads are limited to 256 bits, and writes are limited to 8 to 256 bits in width. Memory can be expanded via paying the cost of gas. However, memory is more costly the larger it grows. The last area is “stack”. The stack is the area where computations are performed. The size of the stack is limited. The maximum size is 1024 elements and contains words of 256 bits. The stack is the cheapest area among all other areas. However, the developer has limited access to the stack.

2.2 Smart Contract

Smart contracts are publicly available programs that run on the Ethereum network. It consists of functions and data, stored at a specific address, 42 character hexadecimal address, on the Ethereum network [26]. The smart contract can send, hold, and receive some amount of virtual money (Ether), and they can interact with another contract on the network. As regular contracts did, smart contracts can define rules. Any user on the network can write, deploy and/or interact by smart contract by executing a defined function on the contract. The code of an Ethereum contract is in a low-level, stack-based bytecode language referred to as Ethereum virtual machine (EVM) code. Users define contracts using high-level programming languages such as Solidity [29], Vyper [18], Serpent [112], Liquidity [50]. There are also low-level programming languages that exist for smart contract implementation, including Rust [62], Yul [50], and Yul+ [50]. The most popular language is Solidity. It is a JavaScript-like language compiled into EVM code using the Solidity compiler.

Listing 2.1 shows an example smart contract that name is *Storage*. The first line represents the compatible compiler versions. For this example, compatible compiler versions are between 0.7.0 and 0.9.0. This is to make sure that the contract does not behave unexpectedly under a different compiler version. The second line shows the name of the contract. In the third line, a state variable is declared called *number* of type *uint256*. The between lines 4 to 6, a function that name is *store* is defined. It is a *internal* function and it accepts an *uint256* as input and does not return value. In the lines between 7 to 9, another function called *retrieve* is declared. This function is *internal* and *view* and returns an *uint256* as an output.

```
1 pragma solidity >=0.7.0 <0.9.0;
2 contract Storage {
3  uint256 number;
4  function store(uint256 num) internal{
5     num = number;
6  }
7  function retrieve() internal view returns (uint256){
8     return number;
9  }
10 }
```

Listing 2.1. An example smart contract

2.2.1 Solidity

Solidity is an object-oriented, high-level language to implement smart contracts. Solidity is a curly-bracket language whose syntax uses curly brackets to enclose blocks. It is similar to programming languages such as C++, JavaScript and it is capable of running on the Ethereum Virtual Machine.

Solidity is a statically typed language and supports inheritance, libraries, and user-defined types. The idea of the Contracts in Solidity is similar to classes in object-oriented languages. Users can declare state variables, functions, function modifiers, events, errors, struct types, and enum types. Solidity supports many of the data types supported by traditional programming languages. Solidity does not use "undefined", or "null" values, and declared variables have a default value that depends on their type.

Functions in the Solidity is an executable code unit. They can be defined inside or outside of contracts. Listing 2.2 represents notation of the function definition. As seen in Listing, a developer should start to function definition with *function* keyword. After that, it will assign the name of the function and function parameters if any. Then, the developer needs to specify the visibility (external, internal, public, private), function modifier if any, and state mutability (pure, view, payable) of the function. Lastly, return variables need to be assigned, if any.

```
1 function NameoftheFunction(<parameter types>) {external|internal|
    public|private} [modifier1|modifier2|...|modifierN] [pure|view|
    payable] [returns (<return types>)]
```

Listing 2.2. Notation of Function Type

The visibility of the function needs to be specified with specified keywords (external, internal, public, private) in Solidity. External functions contain an address and a function signature of themselves. These functions can be passed or returned from external function calls. However, external functions cannot be called from the same contract or inherited contracts. Internal functions can be called inside the contract, internal library functions, and inherited functions. Public functions can be called anywhere in the contract. Private functions only can be called by the contract that is defined.

Function modifier in Solidity can be used to change the behaviour of the functions in a declarative way [29, 115]. A function can use multiple modifiers. Listing 2.3 shows a code snippet for onlyOwner modifier that restricts calling of a function to the owner.

```
1     modifier onlyOwner {
2         require(msg.sender == owner);
```

```
3            _;
4        }
```

Listing 2.3. OnlyOwner modifier

State mutability of the function can be specified with keywords (pure, view, payable) in Solidity. Pure keyword represents that the function neither read nor modify the state. View keyword means that the function can read but cannot modify the state. Lastly, the payable keyword represents that the function can accept ETH sent to the contract.

Solidity has two types of special functions. The first function is "Receive Ether Function". It is a special function, and a contract can have a maximum of one receive function. Listing 2.4 shows the declaration of this function. As seen in Listing 2.4, receive function cannot have arguments and return anything. Visibility of the function must be external, and state mutability must be payable. This function is executed on plain Ether transfers such as .send() or .transfer() to the contract with empty calldata. The other special function is "Fallback Function". As "Receive Ether Function", the fallback function also can be at most one in a contract. Listing 2.5 shows the declaration of fallback functions. The visibility of the fallback function must be external, too. This function is executed in two situations. First, if any of the other functions in the contract do not match the given function signature, the fallback function is executed automatically by contract. The second scenario is, if a call with an empty calldata is received to contract and there is no "Receive Ether Function" in the contract, the fallback function is executed by contract.

```
1 receive() external payable { ... }
```

Listing 2.4. Notation of Receive Ether Function

```
1 fallback () external [payable]{ ... }
```

Listing 2.5. Notation of Fallback function

2.2.2 Well-known vulnerabilities in Smart Contracts

This subsection explains twelve well-known vulnerabilities reported frequently in the smart contract weakness classification (SWC) registry by smart contract users.

Unchecked send: A smart contract is able to call another contract with specified functions such as send, transfer, call etc. Calling an external contract can create some anomalies in the contract. Since, even if the called contract throws an exception,

execution in the contract will not stop. If the call fails accidentally or an attacker forces it to fail, it may cause unexpected behaviour in the subsequent program logic. The best way to handle these unexpected anomalies is to check the call function's return value inside the contract after each call. Listing 2.6 shows a code snippet with unchecked send vulnerability in line 4.

```
1 contract Wallet{
2   ...
3   function cash(uint amount, address _to) public {
4       winner.send(amount);
5   }
6   ...
7 }
```

Listing 2.6. Code snippet with Unchecked send vulnerability

Timestamp dependency: Any operation on the blockchain relies on a timestamp, a smart contract receives a timestamp that specifies the time when the block was generated. A malicious miner could manipulate the timestamp for the generated block for his own purpose. This timestamp dependence vulnerability was exploited in the GovernMental Ponzi scheme [46]. The malicious miner generated a block for his transaction with a modified timestamp that delayed his transaction to be the final one, helping him win the funds from the smart contract. Listing 2.7 shows a code snippet with timestamp dependency vulnerability in line 5.

```
1 contract PonziGame{
2   ...
3   function winner() external payable {
4      require(msg.value == 5 ether);
5       if(now % 15 == 0) {
6          msg.sender.transfer(this.balance);
7       }
8   }
9   ...
10 }
```

Listing 2.7. Code snippet with Timestamp dependency vulnerability

Unprotected SELFDESTRUCT instruction: Selfdestruct instruction is a way to remove the contract from the blockchain network. Developers must be sure that selfdestruct instruction is not accessible by every miner in the network. That is, selfdestruct instruction must be just accessible by authorized users like the contract owner. Due

to missing or insufficient access controls, attackers can remove the contract from the ethereum network. It can lead to frozen fund errors. For instance, In November 2017, one user accidentally froze more than 500,000 ether (worth around $268 million) held in Parity wallet accounts [33]. Listing 2.8 shows an example code snippet with Unprotected SELFDESTRUCT instruction vulnerability in line 8.

```
1 contract WalletLibrary{
2   ...
3   modifier onlymanyowners(bytes32 _operation) {
4     if (confirmAndCheck(_operation))
5       _;
6   }
7   function kill(address _to) onlymanyowners(sha3(msg.data))
        external {
8     selfdestruct(_to);
9   }
10 }
```

Listing 2.8. Code snippet with Unprotected SELFDESTRUCT instruction vulnerability

Authorisation through tx.origin: *tx.origin* is a global variable in Solidity which returns the address of the account that sent the transaction. Using the variable for authorisation could make a contract vulnerable if an authorized account calls a malicious contract. A call could be made to the vulnerable contract that passes the authorisation check since tx.origin returns the original sender of the transaction which in this case is the authorised account. The suggested fix is to use msg.sender for authorisation. Example contracts with this weakness can be found in the Smart Contract Weakness registry [96]. Listing 2.9 shows an example smart contract with Authorisation through tx.origin vulnerability in line 7.

```
1 contract MyContract {
2  address owner;
3  function MyContract() public{
4     owner = msg.sender;
5  }
6  function sendTo(address receiver, uint amount) public{
7     require(tx.origin == owner);
8     receiver.transfer(amount)
9  }
10 }
```

Listing 2.9. Code snippet with Authorisation through tx.origin vulnerability

Reentrancy: Reentrancy occurs when a function invocation is interrupted by another call before completing its work. This vulnerability causes undesired interactions and changes to the shared states of the functions. For example, one of the most well-known attacks, DAO, exploited a reentrancy vulnerability and led to substantial financial losses [10]. The simplest way to prevent this kind of attack is to ensure that all internal work (changes to state) within a function invocation is completed before calling an external function. Listing 2.10 shows an example smart contract with reentrancy vulnerability in lines between 7 to 9.

```
1 contract SimpleDAO {
2   mapping (address => uint) public credit;
3   function donate(address to) payable public{
4     credit[to] += msg.value;
5   }
6   function withdraw(uint amount) public{
7     if (credit[msg.sender]>= amount) {
8       require(msg.sender.call.value(amount)());
9       credit[msg.sender]-=amount;
10    }
11  }
12 }
```

Listing 2.10. Code snippet with Reentrancy vulnerability

Transaction Ordering dependency: An inherent issue in blockchains is that there is no guarantee on the execution order of transactions. While Solidity does not support concurrency, a miner can influence the outcome of a transaction due to its own reordering criteria. It has been observed that miners exploit this to create transactions that are ordered earlier to claim bonuses, like in the context of Initial Coin Offerings [53]. This issue with transaction orderings is seen across blockchains and is not programming language-dependent.

Write to Arbitrary storage location: Data of the smart contract is permanently stored at some storage location on the Ethereum Virtual machine. The contract is responsible for the security of the storage locations (i.e., write and read from storage). That is, the contract must be sure that only authorized users or contract accounts have access to sensitive storage locations. Otherwise, an attacker is able to manipulate sensitive information in the storage by altering the address of the contract owner, balance of an account etc.

Uninitialized storage pointer: Local variables within functions in Solidity are as-

signed as storage or memory depending on their type in default. Uninitialised local storage variables can point to unexpected storage variables in the contract, leading to intentional or unintentional vulnerabilities. Listing 2.11 shows an example code snippet with Uninitialised storage pointer vulnerability in line 9.

```
1 contract CryptoRoulette {
2   ...
3   struct Player{
4    address Paddress;
5    uint256 number;
6   }
7   ...
8   function game(uint256 _num) payable public {
9     Player player;
10    player.Paddress = msg.sender;
11    player.number = _num;
12    ...
13  }
14 }
```

Listing 2.11. Code snippet with Uninitialized storage pointer vulnerability

Delegatecall to untrusted callee: The only difference between delegatecall and call message is that the code at the target address is executed in the context of the calling contract. Each delegate function call is not making the contract vulnerable. It is more related to the function's argument. In other words, if a user sends private information such as msg.data as an argument, this function call can cause vulnerability because an attacker can manipulate arguments with the signature of a function. For example, one of the most well-known deletage call attack happened in the Parity-multisig-wallet contract in June 2017 and led to around $30 million losses [85].

Integer Overflow/Underflow: The Solidity programming language supports unsigned and signed integers with widths ranging from 8 to 256 bits(eg. uint8, uint16). Smart contracts make heavy use of arithmetic operations for computations in transactions. The computation that exceeds the limit of the integer type results in overflow/underflow which deviates from the desired behaviour and is commonly cited as a security vulnerability [58]. Integer overflow was recently detected in a token smart contract (BECToken) that erroneously allows a large amount of tokens to be sent to receiver addresses in its Batch Transfer function. Listing 2.12 shows an example code snippet with integer overflow vulnerability in line 5.

```
1 contract BecToken{
2 ...
3 function batchTransfer(address[] _receivers, uint256 _value)
      public whenNotPaused returns (bool) {
4     uint cnt = _receivers.length;
5     uint256 amount = uint256(cnt) * _value;
6     require(cnt > 0 && cnt <= 20);
7     require(_value > 0 && balances[msg.sender] >= amount);
8     balances[msg.sender] = balances[msg.sender].sub(amount);
9     for (uint i = 0; i < cnt; i++) {
10        balances[_receivers[i]] = balances[_receivers[i]].add(
              _value);
11        Transfer(msg.sender, _receivers[i], _value);
12    }
13    return true;
14  }
15  ...
16 }
```

Listing 2.12. Code snippet with Integer Overflow vulnerability

Division by zero: Division by zero errors is another common cause of undesired behaviour in arithmetic operations in smart contracts. The Solidity compiler can detect division by zero errors only if it can statically determine the divisor is zero. If this is not the case, like when the divisor is data dependent on inputs to the smart contracts, then the Solidity compiler will not be able to catch division by zero vulnerabilities. Developers are expected to manually insert checks for such division by zero cases.

2.3 Software Testing

Software testing is a process to determine whether a system does what it is supposed to do in the correct form [78]. The testing process aims to indicate that the software performs as intended under a range of inputs and checking the outputs of the software are the same as the expected output.

2.3.1 Terminology

Under this section, I would like to provide definitions of terms used in this thesis in order to be consistent and avert confusion.

- **software/system under test (SUT)** - software or system that is tested.
- **software specification** - an explanation of the expected behavior of the system under test.
- **test case** - combination of input data and its expected output.
- **test input** - used input data to run SUT.
- **test suite** - collection of test cases.
- **test output** - the output generated by the SUT after test execution under given test input.
- **expected output** - the correct output data for a specific test input.
- **test result** - A result that indicates whether a test has passed or failed.

```
1  pragma solidity ^0.8.7;
2  contract Lotto{
3  ...
4  function play (uint[] memory bets, uint[] memory playersid)
       public returns(uint[] memory){
5    uint total = 0;
6    uint[] memory winners = new uint[](playersid.length);
7
8    if(bets.length == playersid.length && playersid.length>3){
9        for(uint i=0; i<bets.length;i++){
10           total = total + bets[i];
11       }
12       for(uint i=0; i<bets.length;i++){
13           if(bets[i] == (total % 20)){
14              winners[i] = playersid[i];
15            }
16            else{
17                 winners[i] = 0;
18            }
19        }
20   }
21   return winners;
22 }
23 }
```

Listing 2.13. Solidity Lotto Game Code snippet

2.3.1.1 Example:Lotto Game

To illustrate the concepts in software testing, Listing 2.13 represents the play function that is written in Solidity as a part of the Lotto contract. Software specification of the function as follows:

1. Play function accepts two unsigned integer arrays as inputs.
2. Length of these two arrays must be the same.
3. Length of the playersid array must be bigger than three.
4. Play function returns an array, winners, that size is equal to the size of playerids array.
5. If the remainder of the sum of the numbers in the bets array divided by twenty is equal to any number in the bets array, the player who sent that number is added to the winners array. Otherwise, the function returns the zero-filled array of winners.

Test ID	**Test Input**		**Expected Output**
	bets	playersid	
1	[15,16,2,12,8,9]	[1,2,3,4,5,6]	[0,0,3,0,0,0]
2	[4,17,16,5]	[8,5,7,1]	[0,0,0,0]
3	[7,4,8,3,2,16,4]	[5,1,3,4,8,9,10]	[0,1,0,0,0,0,10]

Table 2.1. Example test suite for the play function in Lotto contract

Table 2.1 shows an example test suite that contains three test cases for the play function in the Lotto contract based on the specification of this function. The expected output for each test input is manually calculated by the developer with respect to specifications.

2.3.2 Software Testing Workflow

Figure 2.1 represents the software testing workflow. As seen in Figure 2.1, software testing workflow consists of three main processes. The first one is *test generation*. The test generation process can be done in two ways - manual writing of test or use of automated techniques. The second process is *test execution*. During this process, each

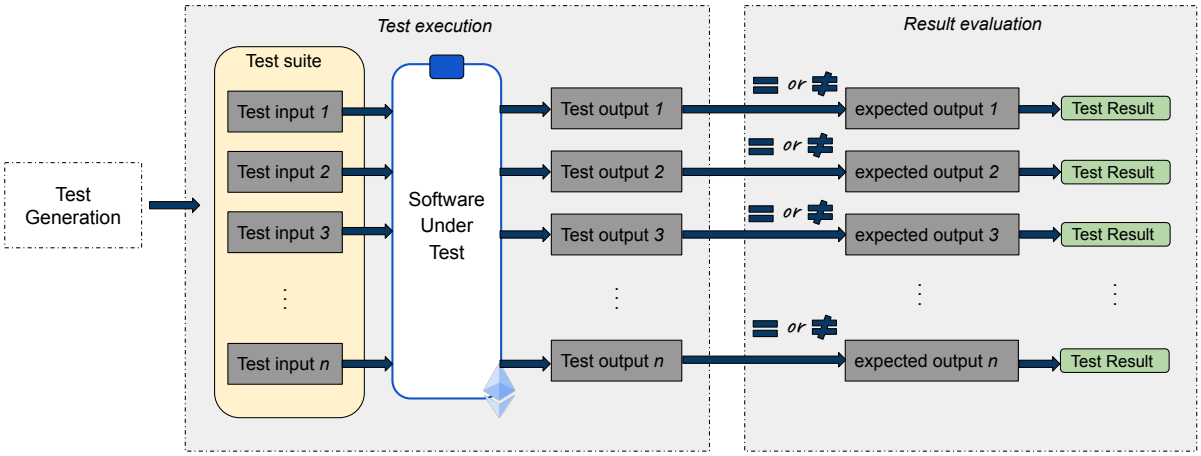

Figure 2.1. Software Testing Workflow

test input generated in the *test generation* process is executed system under test, and test outputs are recorded for each test input. The last process is *result evaluation*. In this process, the recorded test outputs are compared to expected outputs. If the expected and recorded outputs are the same, the test is classified as passed; otherwise, it is classified as failed.

2.3.3 Test Input Generation

As mentioned in Section 2.3.2, the test generation process is the first step of the software testing workflow, and it can be done with manual or automated techniques. However, manually test writing can be very time-consuming, and therefore, it is not desired by the developer and/or the tester. Automated test generation techniques are commonly used in academia and industries. Using automated test generation techniques can widely reduce the time-consuming throughout the test generation process [95]. In literature, many test generation algorithms exist for testing software in various domains [7, 95]. Under this section, we explain the algorithms that are relevant to the scope of this thesis.

2.3.3.0.1 Fuzz Testing. Fuzz testing or simply fuzzing is one of the most popular vulnerability detection techniques [66], and it has proven to be fast and effective in various domains [42, 63, 65, 92]. Fuzzing aims to generate normal or abnormal data as test inputs to discover exceptions in an application [104]. Figure 2.2 shows the workflow of the fuzz testing. The working process of fuzzing starts with the generation of a set of test inputs. After the test input generation process, fuzzing executes the generated fuzz data in the SUT until predefined conditions are satisfied. When the execution ends, it

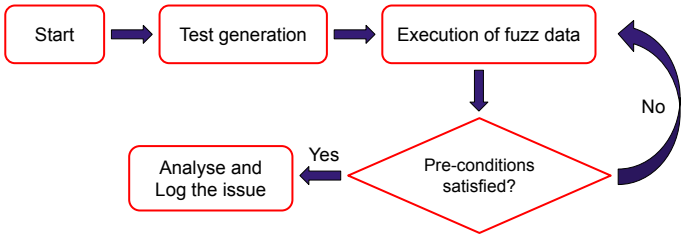

Figure 2.2. Working process of fuzz testing

analyses the system behaviour and logs issues for SUT.

Fuzz testing can be classified as *mutation-based fuzzing* and *generation-based fuzzing*, depending on the initial input generation type [66]. Mutation-based fuzzers are used in many of the big projects such as Windows OS, Linux OS [83, 104]. Mutation-based fuzzing takes a set of valid test inputs as initial. Then, it generates new test inputs by randomly mutating the initial inputs without knowing the input model. For example, a file format fuzzer takes a valid file sample as a seed; it then modifies the original file and tests the file format. The main advantage of the mutation-based technique is that this technique has not need knowledge about the specifications of the SUT. On the other hand, the existence or quality of a corpus depends on the seed input [104]. The latter technique, generation-based fuzzing, takes an input model as seed and generates new inputs based on given model [76]. For example, a graphical user interface (GUI) fuzzer takes the specifications or protocols of the GUI as an input model, then generates syntactically valid random input data to test the GUI. According to the research done by Miller et al.(2007), [76], generation-based fuzzing performed up to 76% better than mutation-based fuzzing techniques.

2.3.3.0.2 Genetic Algorithm. Genetic Algorithms (GA) are metaheuristic algorithms that imitate natural evolution to produce effective solutions for optimization and search problems [114]. Genetic algorithms have proven effective for testing Java programs [38], mobile [13, 72, 74], web [6, 30] and Internet of Things (IoT) applications [9] among others. Implementation of a GA has five main phases. The first phase starts with a defined number of individuals called a population. Each individual in the population is a solution for the problem algorithm wants to solve. The second phase is determining the Fitness function and calculating the fitness score for each individual in the population. The fitness function typically used in test generation are maximizing coverage, minimizing execution time (cost), and is to calculate the fitness score for each individual with respect to the ability of an individual to combat, among others.

The third phase is the selection; the idea of this phase is to pick the best individuals based on fitness scores and pass their genes to the next generation. The next phase is crossover. The crossover operator is a convergence operation intended to pull the population towards a local maxima. Crossover is a process of taking two existing individuals (parents) and producing a new individual by combining them (offspring). The final phase is mutation. The mutation operator modifies one or more test inputs in an individual according to a given probability. The goal of the mutation process is to increase the diversity of the population and to reduce similarity between individuals. After the last phase, the algorithm terminates if any stopping conditions are satisfied; otherwise, it repeats the phases between two to five until one of the stopping conditions is satisfied.

2.3.4 Test Effectiveness

Test effectiveness is a way to understand to how extent testing is done or to what extent the goal is accomplished. The effectiveness of testing has been traditionally measured in terms of code coverage achieved and fault finding [45].

2.3.4.1 Code Coverage

Code coverage determines how much of the SUT is tested by given test input. Different coverage criteria can be used for different interests to measure code coverage [24]. In the literature, many coverage criteria can be found [45, 119] such as statement coverage, branch coverage, condition coverage, etc. The coverage criteria used in our research are explained in section 6.2.2.2.

- **Statement coverage** reports the ratio of statements executed by test inputs to total number of statements in the source code.

$$Statement\ Coverage\ = \frac{\#Executed\ statements}{Total\ \#statements\ in\ the\ source\ code} \tag{2.1}$$

- **Branch coverage** measures the fraction of the branches covered by the test inputs to total number of branches in the source code.

$$Branch\ Coverage\ = \frac{\#Covered\ branches}{Total\ \#branches\ in\ the\ source\ code} \tag{2.2}$$

- **Condition coverage** measures the fraction of the conditions in decision expression that have been evaluated to both true and false.

$$Condition\ Coverage\ = \frac{\#Executed\ operands}{Total\ \#operands\ in\ the\ source\ code} \tag{2.3}$$

2.3.4.2 Mutation Testing

Mutation testing is a fault-based testing technique that modifies the source of the SUT and injects errors. This technique is commonly used to determine the effectiveness of the test cases in many domain [59, 86, 99]. The idea of mutation testing is to execute the modified copy of the SUT, which is called a mutant, with the test suite and check whether test cases are able to detect these injected errors. One of the outcomes of this testing method is the mutation score that represents the ratio of the number of the revealed mutant over the total number of the mutant [55] - see Formula 2.4.

$$Mutation\ Score\ = \frac{\#killed\ mutants}{Total\ \#mutants} \tag{2.4}$$

Many mutation operations have been proposed by researchers, as an example, some of them are listed as follow:

- **Statement deletion** - delete one of the statement in the SUT.
- **Statement duplication** - duplicate one of the statement in the SUT.
- **Constant replacement** - replace one of the constant value by another value in the SUT.
- **Operator replacement** - replace one of the operator with another one in the SUT.

```
1 for(uint = 0; i<bets.length;i++){
```

Listing 2.14. Original Statement

```
1 for(uint = 0; i<bets.length;){ /* i++ statement is deleted. */
```

Listing 2.15. Mutant for statement deletion

```
1 for(uint = 0; i<bets.length;i++){ i++; /* i++ statement is
    duplicated. */
```

Listing 2.16. Mutant for statement duplication

```
1 for(uint = 1; i<bets.length;i++){ /* uint = 0 is changed to uint =
      1 */
```

Listing 2.17. Mutant for constant replacement

```
1 for(uint = 0; i< = bets.length;i++){ /* < operator is changed to
     <= */
```

Listing 2.18. Mutant for operator replacement

Chapter 3

Related Work

This chapter presents a literature review of the existing work related to the contributions of the thesis. In Section 3.1, we explain the existing Solidity code instrumentation tools for Solidity smart contracts in literature. The following section presents analysis tools for Solidity smart contracts grouped by technique. Lastly, in Section 3.3, we describe the existing testing tools for smart contracts, and Solidity test effectiveness measurement works in terms of code coverage and fault finding ability.

3.1 Code instrumentation frameworks for Solidity Smart contracts

With the increased use of smart contracts across application domains, there is a crucial need for a unified framework that supports and facilitates Solidity code analysis, understanding, transformation, and development of tools for verification and testing that provide strong security guarantees. Traditional programming languages are generally supported by a comprehensive framework for code instrumentation, monitoring, optimisation and code generation such as LLVM/Clang [64] for C/C++. LLVM/Clang framework support is lacking for smart contract programming languages like Solidity.

To our knowledge, there is no single unifying framework for analysis, instrumentation, optimisation and code generation of Solidity contracts at the source code level. Our framework, SIF, in this thesis is the first in the domain.

Table 3.1. Summary of Existing Static Analysis Tools

Tool	Open-Source	Limitations
SolidityCheck	✓	False-positive Not applicable for all Solidity versions
SmartCheck	✓	False-positive
Slither	✓	
Securify	✓	
Clairvoyance	✗	
Zeus	✗	False-positive No support for all construct in Solidity
Vandal	✓	False-positive Limited vulnerability support
MadMax	✓	
VeriSmart	✓	Limited vulnerability support

3.2 Analysis tools for Solidity Smart Contracts

In the last few years, several analysis tools for detecting vulnerabilities in smart contracts have been proposed. We analysed seventeen different existing analysis tools in the literature. Nine of them rely on static analysis techniques, and eight of them rely on symbolic execution techniques.

3.2.1 Static Analysis

Static analysis is a method to analyse the program without actually executing it. This method can be performed at the source code or bytecode levels and analyse the entire code block. The primary drawback of the static analysis technique is that they tend to produce a high number of false positives [94, 97]. In addition to a high false-positive rate, Static analysis techniques in the smart contract field face additional drawbacks such as lack of certain support for commonly used constructs and high time complexity. Table 3.1 presents the summary of the existing static analysis tool in the field.

SolidityCheck [122] retrieves user-selected functions in the contract at the source

code level. Then, to simplify the selected functions, it extracts statements according to predefined different keywords using regular expressions and compares the extracted statements with predefined vulnerability patterns. However, doing simplification in the source code with a regular expression is not convenient and applicable in general. Like other programming language developers, smart contract developers also do not follow a template to implement contracts. Therefore, it is hard to apply simplification in the source code level with predefined regular expressions. Moreover, Solidity is a developing language; it is not mature enough like Java, C, C++, etc. To improve the language, Solidity developers release new versions frequently on short notice; many of the versions come with new keywords and constructs. Thus, predefined regular expressions will not be capable of covering these constructs and keywords.

SmartCheck [108] translates Solidity code into an XML-based intermediate representation for analysis. It then uses XPath queries on the intermediate representation to detect vulnerability patterns. Slither [37] translate Solidity smart contracts into an intermediate representation called SlithIR and converts the intermediate representation into Static Single Assignment (SSA) form. Then, it symbolically executes this form and tracks the data dependency using tainted tracking technique.

Securify [111] uses abstract interpretation to check violation patterns or compliance properties on the semantic extracted from the smart contract. To detect the reentrancy vulnerabilities in smart contracts, Clairvoyance [120] generates a cross-contract call graph and control flow graph. Then, it identifies suspicious objects or addresses to track based on usage and locations from generated graphs. Zeus [58] uses abstract interpretation and symbolic model checking to analyse Solidity contracts. This tool does not support commonly used constructs such as Throw, Self-destruct, virtual functions, assembly code block in Solidity.

Vandal [16] translates low-level EVM bytecode to logic relations. It then checks the presence of vulnerabilities in translated logic relations. MadMax [47] built on the top of the Vandal and works on EVM bytecode level. This tool only focuses on gas-related vulnerabilities such as gas costly loops.

VeriSmart [103] takes a Solidity smart contract as an input, and it analyses transaction variables to check the safety of arithmetic operations. This tool, instead of focuses a selection of vulnerabilities in smart contracts, only focuses on integer overflow and underflow vulnerabilities.

Table 3.2. Summary of Existing Symbolic Execution Tools

Tool	Open-Source	Limitations
Oyente	✓	Path explosion False-positive
Maian	✓	False-positive
Osiris	✓	Limited vulnerability support
DefectChecker	✗	Covered defects already supported by Solidity compiler 0.4.20 onward
SCompile	✗	Focuses on monetary transactions
Manticore	✓	Long execution time
Mythril	✓	Time-out
VerX	✓	Need knowledge for Solidity language Need to write predicates for tested contracts

3.2.2 Symbolic Execution

During the program analysis, symbolic execution uses symbolic values to feed the program rather than specific values. This method gathers the path constraints and then uses a constraint solver to produce inputs that trigger each branch [20, 61]. Although symbolic execution can synchronously explore multiple paths that a program can take, it faces problems such as path explosion, time complexity [20, 97]. Also, many of the symbolic analysis tools in existing work uses Z3 solver to solve constraints in smart contracts. However, according to our experiments (see Section 6.3), the Z3 solver does not handle Solidity specific and block related constraints (block number, contract balance, etc.). Thus, traditional solvers are not very efficient for smart contracts regarding code coverage and fault-finding ability. Table 3.2 presents the summary of the existing symbolic execution tools in the smart contract domain.

Oyente [71] operates at the EVM bytecode level to detect some known security issues like integer overflow, integer underflow, timestamp dependency, reentrancy. It generates Control Flow Graph (CFG) and gathers path conditions using Depth First Search (DFS), then uses a Z3 solver to generate input to satisfy the gathered conditions. Apart from the path explosion problem, Oyente also suffers from a high false-positive rate for all supported security issues [11, 75].

Maian [82], and Osiris [110] are implemented top of the Oyente tool. Maian takes

the EVM bytecode as an input and uses symbolic execution and concrete validation techniques to detect unchecked send, freezing ether, and unchecked self-destruct vulnerabilities. According to our experiments (see Section 5.4.3), Maian is effective at detecting unchecked send vulnerability compared to other tools. However, it has limited support for well-known vulnerability types. Osiris combines symbolic execution and taint analysis to detect integer overflow and underflow bugs in Solidity smart contracts. It generates CFG from EVM bytecode and symbolically executes the reachable paths of the contract.

Defectchecker [23] takes EVM bytecode as input and detects eight contract defects that cause unexpected behaviours in smart contracts. However, many of the detect they are checking is already covered by Solidity compilers 0.4.20 onwards.

sCompile [21] takes Solidity source code as an input and constructs CFG, and gather all possible paths. To address the path explosion problem, this tool filters the paths as ”critical or noncritical” according to involving monetary transactions, then only works on paths specified as critical. However, many common vulnerabilities such as tx.origin, timestamp usage, integer overflow/underflow, etc. in smart contracts are not related to monetary transactions.

Manticore [77] is a vulnerability detection tool for smart contracts. It takes Solidity source code as an input and then generates CFG to gather path conditions. It uses Z3 SMT solver to solve the constraints. According to our experiments (see Section 5.4.3), the execution time of this tool is significantly slow, even for small and simple contracts. In addition to that, because of the long execution time, this tool faces with time-out problem for the middle (100-150 LOC) and large size (200+ LOC) contracts [32].

Mythril uses symbolic execution, SMT solver and taint analysis to detect security vulnerabilities in smart contracts. According to our experiments (see Section 5.4.3) and other research papers [44, 51], Mythril faces a high number of false-positive and false negatives. Another drawback of this tool is slow analysis time. According to our experiments, Mythril takes 15 mins to analyse this contract, even if a simple and small contract (65 LOC).

VerX [91] combines symbolic execution and abstract interpretation techniques to detect vulnerabilities in a smart contract. It takes Solidity source code and security requirements as inputs and returns the predicate of whether the contract meets the given properties. However, writing security requirements of a smart contract needs knowledge for the Solidity language and structure of the smart contract and is time-consuming.

Table 3.3. Summary of Existing Testing Tools

Tool	Open-Source	Limitations
Echidna	✓	Based on user-provided predicates or test functions
sFuzz	✓	
ContractFuzzer	✓	High rate of false alarms
Harvey	✗	Source code is not accessible Exercised under a tiny dataset by authors The effectiveness of the tool is still a question
CONFUZZIUS	✗	Source code is not accessible The effectiveness of the tool is still a question
ContractWard	✗	Training set is generated based on Oyente result
S-gram	✗	

3.3 Testing tools for Solidity Smart Contracts

Because of the stated limitations in static analysis and symbolic execution techniques in the smart contract field, testing techniques have been emerged to detect security vulnerabilities in smart contracts in the last couple of years. In this section, we present existing test input generation tools for smart contracts in the literature. Table 3.3 presents the summary of the existing testing tools in the field.

Echidna [48] is a Haskell library designed for property-based testing of EVM code. It uses grammar-based fuzzing to generate tests based on user-provided predicates or test functions. However, writing the predicates and test functions requires significant expertise and is time-consuming.

sFuzz [81] combines fuzzing strategy in American Fuzzy Lop [121] (AFL) with a search-based technique for selecting seeds. The objective used in selecting the seeds in a quantitative measure(distance) on how far a seed is from covering any just-missed branch (one control flow edge away from covered node). The technique evolves the initial random population by selecting test cases that cover a new branch and selecting one test case for each just-missed branch that is closest according to their distance metric. sFuzz keeps the number of test inputs generated small by only keeping the best seed for each just-missed branch. Overhead is reduced by only computing distance to just-missed branches, not all uncovered branches. We evaluate this tool under two different datasets - contains 1775 smart contracts (see Chapter 6) and our experiments show that sFuzz is better than other compared tools with respect to code coverage, fault

finding ability, and overhead time. However, code coverage and fault finding ability are still low in any domain considered for that matter.

ContractFuzzer [56] takes Application Binary Interface (ABI) and bytecode files generated by the Solidity compiler as inputs and generates inputs using blackbox fuzzing to detect security vulnerabilities in smart contracts. This tool supports exception disorder, reentrancy, timestamp dependency, dangerous delegate call and freezing ether vulnerabilities. The approach of this tool is similar to our blackbox fuzzing input generator tool which is a part of our property assertion framework. However, there are some vulnerabilities (gas-related vulnerabilities, tx.origin usage, division by zero), that are supported by our tool, are not supported by ContractFuzzer. Also, ContractFuzzer is prone to a high rate of false alarms (or false positive if positive label is used for absence of bugs) for certain types of vulnerabilities such as timestamp dependency (62% reported in their paper).

Harvey [118] is an industrial fuzzing tool to detect some known security vulnerabilities in smart contracts. It generates random inputs, then mutates program inputs to improve coverage on program paths. The main drawback of the tool is its effectiveness is still in question. The source code of the Harvey is not available in public, and the effectiveness of the tool has been examined under a tiny dataset – contains 27 smart contracts.

CONFUZZIUS [109] is a fuzzing based approach to detect some vulnerabilities in smart contracts. It takes the source code as an input and applies an evolutionary-based fuzzing engine to generate inputs. This tool follows traditional genetic algorithm steps - selection, crossover, and mutation. The approach of this tool is similar to our GA implementation. However, our GA implementation and CONFUZZIUS have different selection and mutation criteria. The first difference is that while our GA implementation uses opcode coverage as fitness score, CONFUZZIUS uses combinations of branch coverage and number of storage operations as a fitness score. The second difference is that while our implementation uses a predefined crossover rate to combine two input sequences, CONFUZZIUS uses the order of the write and read sequences in input sequences. It is worth stating that our GA implementation was the first in the smart contract domain when we published our research paper.

ContractWard and S-gram [69] are machine learning based vulnerability prediction tools. ContractWard [113] detects six types of vulnerabilities in smart contracts with machine learning techniques. Then, it extracts the static characteristics of the contracts. After that, it employs five machine learning algorithms to decide whether

tested contracts are vulnerable or not. The machine learning algorithms ContractWard used are namely - eXtreme Gradient Boosting (XGBoost) [25], adaptive boosting (AdaBoost) [40], Random Forest (RF) [15], Support Vector Machine (SVM) [105] and k-Nearest Neighbor (KNN) [28]. S-gram combines the N-gram language modelling and lightweight static semantic labelling to predict potential vulnerabilities by identifying irregular token sequences and optimizing existing in-depth analyzers. The training set of both tools are generated by Oyente. That means, ContractWard and S-gram label the contracts as vulnerable or safe according to the result of Oyente in the training set. However, as mentioned in our related work and our experiments (see Section 5.4.3), Oyente is prone to high false positives. That means, many of the vulnerabilities Oyente raised, actually is not visible in the contract. Therefore, the correctness of the training set of this tool is not strong enough to decide the vulnerabilities in the contract.

3.3.1 Test Effectiveness Measurement

Effectiveness of testing has been traditionally measured in terms of (i) code coverage achieved, and (ii) fault finding [43, 89]. There is limited existing work defining and measuring Solidity specific code coverage and fault finding.

3.3.1.1 Code coverage

We found three pieces of work – (1) Path coverage measurement proposed by Fu et al. [41] in 2019, (2) Statement and branch coverage measurement by Brownie [17], (3) Statement coverage implemented by solidity-coverage [98]. Fu et al. measure path coverage for symbolic execution by gathering sequences of opcode and branch conditions. Measuring path coverage incurs high overhead and their implementation is not easy to use with tests from other techniques. Brownie and solidity-coverage tools assess statement or opcode coverage (and branch coverage with Brownie) by instrumenting the abstract syntax tree followed by analysing the execution trace to detect the opcodes and branches executed. Our approach for coverage measurement is similar but we measure call coverage and event coverage in addition to opcode and branch coverage.

3.3.1.2 Fault finding

We found two tools – (1) Eth-mutants [14] and (2) MuSC [67]. Eth-mutants [14] is an open-source mutant generator for Solidity contracts. However, the mutation

types supported are limited. It only performs replacement of $<=$ to $<$ and $>=$ to $>$. MuSC [67] is another mutation tool that supports a wider range of mutation types, including ones specifically targeting Solidity syntax. This mutation tool operate at the AST level. Approach of our mutation generator tool, MuContract [3] is similar to MuSC. The primary difference between them is that MuSC makes every possible change in the AST for a given mutation type. For instance, for a relational operator mutation type, if there are 100 possible changes of this type in the original contract, then MuSC will generate 100 mutants covering all locations and options. MuContract, on the other hand, selects one change randomly among all possibilities of a relational operator change in the AST. It then generates one mutant for that mutation type. Our Mutant generator, MuContract, was the first in smart contract field.

Chapter 4

SIF: A Framework for Solidity Contract Instrumentation and Analysis

4.1 Introduction

The research contributions of this chapter is the design and implementation of the our instrumentation and analysis framework for Solidity smart contracts. This chapter describes the approach, the implementation and evaluation of the framework using 1838 real smart contracts deployed on the Ethereum network.

During the implementation and evaluation of the framework, we have collaborated with one of our colleagues, another Ph.D. student in our research group. For this research, I implemented 4 out of 7 of the tools. These are Function Listing, SIF Rename, Fault Seeder, and Assertion Analyser. The rest of the tools were implemented by my colleague. The evaluation of the framework has been done by me and my colleague, Chao Peng.

We present SIF, a comprehensive framework for Solidity contract analysis, query,

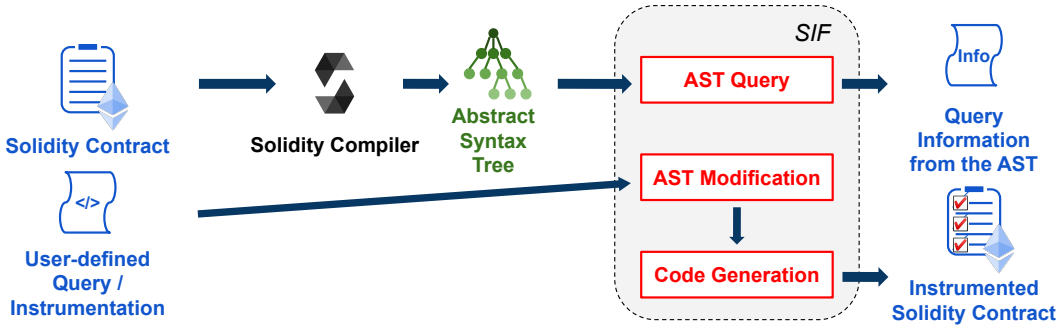

Figure 4.1. SIF work flow

instrumentation, and code generation. SIF provides support for Solidity contract developers and testers to build source level techniques for analysis, understanding, diagnostics, optimisations and code generation.The work flow of the framework is illustrated in Figure 4.1.

SIF has three main capabilities.

1. Provide an interface for users to query the Abstact Syntax Tree (AST) of Solidity code.

2. Support code instrumentation or transformation using pre-defined helper functions that can modify the AST.

3. Support Solidity code generation from AST.

We demonstrate the general purpose nature of the SIF framework in supporting code instrumentation and analysis by using it to implement 7 different tools or utilities for Solidity contracts. Four of these tools query the Solidity abstract syntax tree (AST) to provide information on the contract code and three others instrument and modify the code using the AST. A brief description of the tools is provided below,

- *Function Listing* lists all the function definitions with function names, return lists, parameter lists and which contract they belong to. Useful in summarising and reviewing Solidity contracts.

- *AST Diff* is a syntactic diff tool for comparing Solifity contracts at the AST level. It ignores comments, white spaces and blank lines.

- *Call Graph Generator* produces function call graphs depicting calling relationships between functions in a Solidity contract. Nodes represent functions and edges represent the calls relation. Useful in understanding and reviewing smart contracts.

- *Control Flow Graph Generator* produces a graphical representation of control flow within a Solidity contract. Control flow graphs are very useful in static analysis and program optimisations.

- *SIF Rename* allows the user to easily rename existing identifiers. All definitions and references to the specified identifier will be changed to the new name provided.

- *Fault Seeder* allows developer to artificially seed bugs in smart contracts. The tool currently supports 3 common bug/vunerability types. The seeded bugs can be used to assess effectiveness of testing or verification tools in uncovering vulnerabilties.

- *Assertion Analyser* examines all AST nodes of the smart contract and determines if the node is susceptible to vulnerabilities, such as division by zero, overflow and underflow. Assertions will be inserted by the tool at contract locations susceptible to these vulnerabilties. Assertions will be checked during execution of the modified contract.

We evaluate SIF and the 7 tools built on top of it on a collection of 1838 real smart contracts that are running on the Ethereum network as presented in Section 4.4. Our results show that SIF and the utility tools are easy to use, highly automated and new tools can be easily implemented with the helper functions.

The rest of this chapter is organised as follows. Section 4.2 presents the instrumentation and analysis framework and its implementation. Section 4.3 presents the experiment setup and research questions of our evaluation. Section 4.4 presents the experimental results in the context of the research questions. Finally, Section 4.5 concludes this chapter.

4.2 Approach

4.2.1 SIF Overview

In this section, we present our generic framework, SIF, implemented in C++. SIF enables Solidity code query and instrumentation and works at the AST level. The framework also provides the capability for generating Solidity code back from the AST. SIF supports the entire Solidity syntax, up to version 0.5.3. The workflow of SIF is illustrated in Figure 4.1. SIF starts from the AST of Solidity code, produced by the Solidity compiler. It then accepts user instructions on queries and/or modifications needed. The framework then gathers the desired query information or performs modifications to the AST and finally generates Solidity code from the AST.

In the rest of this Section, we describe the design SIF, how it can be used for query and instrumentation along with instructions needed from the user, a concrete illustration of the query and instrumentation capabilities using seven tools that we

built on top of SIF. The framework with its source code and user guide is available at `https://github.com/sefaakca/SIF`. To allow users to try the framework without having to download and build the source code, we have provided an online version of SIF at `https://wandbox.org/permlink/PnaL6bO9zipKRuKu`.

4.2.2 SIF Design

Operations of SIF are dividied into 3 phases. Phase 1 focuses on representing AST nodes as C++ classes with methods to retrieve and modify information of the node. Phase 2 interacts with user defined query and/or instrumentation functions, and traverses the AST to perform the desired operation. Phase 3 generates Solidity code from the AST. We discuss each of these phases in more detail in the rest of this Section. We use an example `struct definition` named `Request` shown in Listing 4.1, containing a data element and a method, to illustrate the different phases.

```
1 struct Request {
2   bytes data;
3   function (bytes memory) external callback;
4 }
```

Listing 4.1. Struct example

Phase 1: AST Representation. The Solidity compiler generates the AST from Solidity code in two formats: plain text and JSON (JavaScript Object Notation, a structured data format). In an attempt to make our tool for code instrumentation generic and easy to use, we use C++ classes as our intermediate AST representation. Given the Solidity AST, SIF first traverses the AST and for each node, it instantiates a class of that node type with all the associated information. Figure 4.2 shows a struct definition appearing in an example AST. The classes contain information about the AST node in their data fields, and provide methods to query and modify the data. Listing 4.2 shows the C++ class representation of a struct definition node in the AST. The C++ class provides methods for getting name of the struct, setting it to a different name, querying number of fields in the struct, getting a particular field, adding, removing and updating a field. These methods facilitate query and modification of the AST node.

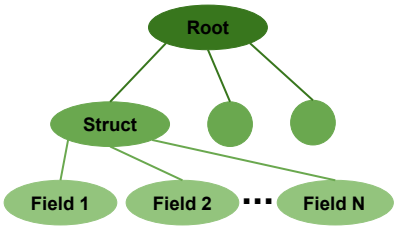

Figure 4.2. An AST example highlighting a struct definition

```
1 class StructDefinitionNode : public ASTNode {
2 public:
3   StructDefinitionNode() : ASTNode(); //Constructor
4   string source_code(); //produce source code
5   string get_name(); //get the name
6   void set_name(new_name); //update the name
7   int num_field(); //get the number of fields
8   void add_field(new_field); //add a new field
9   void remove_field(index); //remove a field
10  void update_field(index, _new); //update a field
11  ASTNode get_field(index); //get a field by
12 private:
13  string name; //name of the struct
14  vector<ASTNode> fields; //list of struct fields
15 };
```

Listing 4.2. Representing structs using a C++ class

Phase 2: Query and Instrumentation. SIF interacts with users for AST query and instrumentation through a function *visit* that it declares. SIF traverses the AST starting from the root node in a depth-first fashion using the *visit* function. The user can implement queries and modifications, if any, that are needed within the *visit* function. For example, the user can change the name of the example struct *Request* from Listing 4.1 to *DirectRequest* by implementing it in the *visit* function, as shown in Listing 4.3. Each time SIF visits an AST node, the *visit* function is called to process operations defined by the user. In this example, the *visit* function first determines whether the current node is a `struct definition` and then checks whether the struct name matches *Request*. If there is a match, the struct name is changed to *DirectRequest*. The methods, *get_name() and set_name()*, used in this implementation are helper methods provided by SIF to facilitate writing of AST queries and instrumentations.

```
1 void visit(ASTNode* node) {
2   if (node->get_node_type() == StructDefinition) {
```

```
3     StructDefinition* sd = (StructDefinition*) node;
4     if (sd.get_name() == "Request")
5       sd.set_name("DirectRequest");
6   }
7 }
```

Listing 4.3. Change the name of a struct in SIF

Phase 3: Solidity code generation. For each type of AST node classes, SIF defines code templates that allow corresponding Solidity code to be generated. Listing 4.4 illustrates the code template for the `StructDefinitionNode`. In the listing below, the template generates Solidity code for a `struct definition` by first printing the `struct` keyword, followed by the name of the struct, a left brace indicating the start of struct element definitions. The template then iterates through the list of struct element definitions. Once an element definition is visited, the source code of that definition is appended to the struct source code. Once all the element definitions are completed, a right closing brace is added, indicating the end of the struct definition. If the name of the struct were changed by the user, as shown in Listing 4.3, the new struct name will appear in the generated Solidity code.

```
1 string StructDefinitionNode::source_code() {
2   string source = "struct " + name + "{\n";
3   Iterate the list of element definitions:
4     source += element.source_code() + ";\n";
5   return source + "}\n";
6 }
```

Listing 4.4. Source code template of structs

4.2.3 Using SIF

SIF provides an interface in the form of predefined functions - *before*, *visit* and *after*, to perform user-defined AST queries and instrumentations. We have presented in Section 4.2.2 how the *visit* function can be used for this purpose. The *before* function is called by SIF in advance of traversing the AST, and can be used for data initialisation. The *after* function is called when visiting the AST is completed and helps summarise information gathered from the AST. In the following Section, we demonstrate how these three functions can be used in custom utility tools built with SIF.

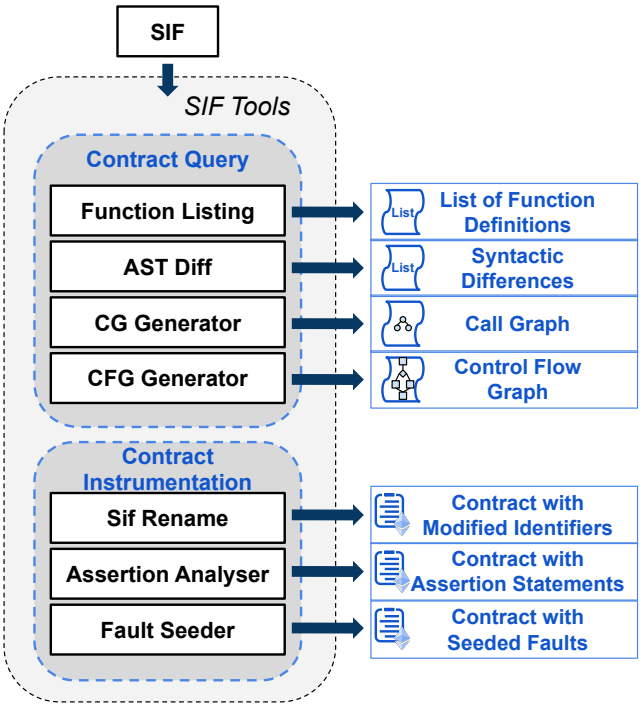

Figure 4.3. Using SIF to build tools

4.2.4 Tools using SIF

To illustrate the framework's extent of support for building custom queries and instrumentations, we implemented seven utility tools for Solidity contracts, as shown in Figure 4.3. The seven tools were inspired by tools built over other mature code instrumentation frameworks, such as Clang LibTooling for C/C++. We discuss and present each of these tools in the rest of this Section.

1-) Function Listing: Given a Solidity file with multiple contracts and function definitions, this tool outputs a list of functions that appears in the file with function names, parameter lists, return lists and the contracts in which they are defined. It is worth noting that in Solidity, functions are considered members of contracts and appear as children of contract definition nodes in the AST representation. To implement the `Function Listing` tool, we use the pre-defined *visit* function within SIF. We check whether the visited node is a contract definition node. For contract definition nodes, we check all the children nodes to see if they are of type function definition. If function definition nodes exist, we record information on its name, parameters, and return values using SIF's helper methods for this node type. Once SIF finishes traversing the AST, the recorded information on functions definitions is printed. Listing 4.5 shows the list

of function definitions printed by this tool for the AztraToken contract in Etherscan[1].

```
1 [In AztraToken] AztraToken() returns ()
2 [In AztraToken] _transfer(address _from, address _to, uint _value)
    returns ()
3 [In AztraToken] transfer(address _to, uint256 _value) returns ()
4 [In AztraToken] transferFrom(address _from, address _to, uint256
    _value) returns (bool success)
5 [In AztraToken] burn(uint256 _value) returns (bool success)
6 [In AztraToken] approve(address _spender, uint256 _value) returns
    (bool success)
7 [In AztraToken] transferOwnership(address newOwner) returns ()
8 [In AztraToken] burnFrom(address _from, uint256 _value) returns (
    bool success)
9 [In AztraToken] mintToken(address target, uint256 mintedAmount)
    returns ()
10 [In AztraToken] freezeAccount(address target, bool freeze) returns
    ()
11 [In AztraToken] transferOwnership(address newOwner) returns ()
```

Listing 4.5. Function definitions summerised by Function Listing

2-) AST Diff: takes two Solidity contracts and compares them at the AST level to report syntactic differences. The tool ignores differences in comments, white spaces, and empty lines. Implementation using the *visit* function starts from the root node of one smart contract. Every node visited within this smart contract is compared with nodes in the other contract. Differences in data fields within the node are reported. Extra or missing nodes are also shown.

3-) CG Generator: illustrates the calling relationships between functions as a call graph (CG). The tool is implemented using the *visit* function first to process two types of AST nodes: function definition and function call. The tool maintains a map containing the callee function with the caller functions it is associated with. After traversing the full AST, the graph is drawn by the *after* function using a graph drawing tool, Graphviz [34], with function names as nodes and edges based on the relations recorded in the map. Figure 4.4 presents the call graph generated by our tool for the AztraToken contract.

[1]The contract AztraToken is available at `https://etherscan.io/address/0x6962E259a8f9633C4494764628A7984cCEd58e10`

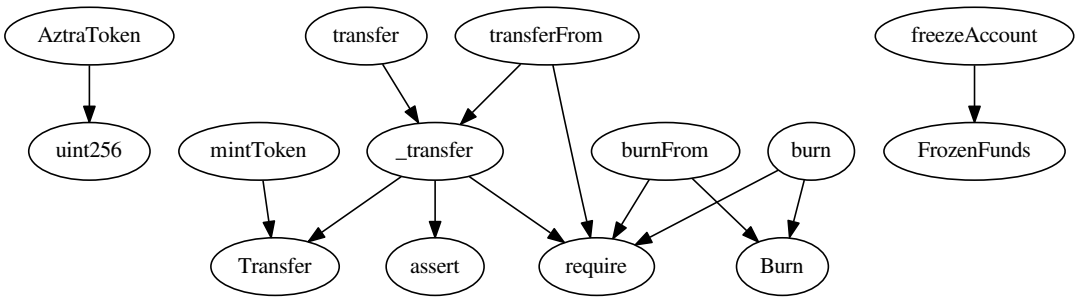

Figure 4.4. Call graph of the smart contract AztraToken

4-) CFG Generator: Control flow graphs (CFG) are used to illustrate control flow within a program [5] and is fundamental to many static analysis and compiler optimisation techniques. The tool `CFG Generator` produces control flow graphs for functions or subroutines within Solidity contracts.

```
1 function uint2str(uint i) internal pure returns (string) {
2   if (i == 0) return "0";
3   uint j = i; uint len;
4   while (j != 0){
5     len++;
6     j /= 10;
7   }
8   bytes memory bstr = new bytes(len);
9   uint k = len - 1;
10  while (i != 0){
11    bstr[k--] = byte(48 + i % 10);
12    i /= 10;
13  }
14  return string(bstr);
15 }
```

Listing 4.6. Function to generate a control flow graph using CFG Generator

Figure 4.5 shows the CFG generated by our tool for the *uint2str* function, shown in Listing 4.6, that converts an unsigned integer variable to a string. The *uint2str* function is found in contract *Item* from Etherscan[2].

[2]The contract Item is available at `https://etherscan.io/address/0x5f896c654a08323dbe16aded331c461ccaeeb370`

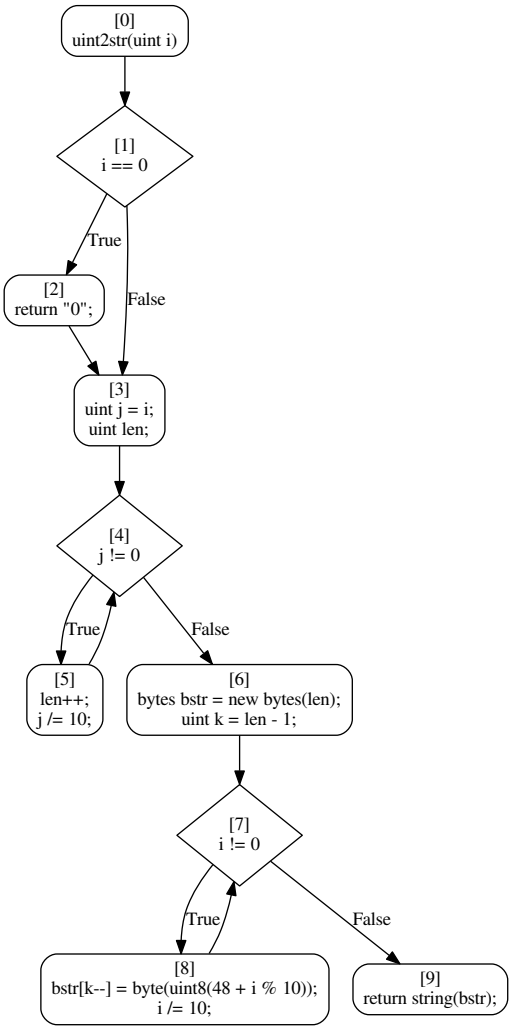

Figure 4.5. Control flow graph of the function uint2str in the smart contract Item

`CFG Generator` traverses the AST and clusters nodes in a straight line code sequence into basic blocks. When a node introducing control flow is encountered, the most recent basic block ends and is linked to the condition node with the control flow. The true branch of this control flow is linked to the first basic block of the then branch for an if statement or the loop body if it is a loop. The false branch is linked to the first basic block of the else branch for an if statement or the first basic block immediately after the loop body if it is a loop. Basic blocks and links between them are maintained and recorded in a data structure as the AST is traversed. Once the AST traversal is completed, the *visit* function passes the data structure to Graphviz [34] to generate a graph depicting basic blocks and control flow. The data structure and resulting CFG is shown in the Listing 4.7 and Figure 4.5, respectively.

Table 4.1. Vulnerability Types

Type of Vulnerability	Assertions Added by Assertion Analyser	Fault seeded by Fault Seeder
Division by Zero	require(c != 0); a = b / c;	Statement with division by zero is inserted.
Unsigned Overflow	a = b + c; assert(a >= b && a >= c);	Arithmetic operation resulting in overflow is inserted.
	a = b * c; (b != 0 && c != 0)? assert(a >=b && a >= c): assert(a == 0);	
Unsigned Underflow	a = b - c; assert(b >= a && b >= c);	Arithmetic operation resulting in underflow is inserted.
Signed Overflow / Underflow	a = b + c; assert((c >= 0 && a >= b) || (c < 0 && a < b));	Arithmetic operation resulting in overflow / underflow is inserted.
	a = b - c; assert((c >= 0 && a <= b) || (c < 0 && a > b));	
	a = b * c; (b != 0 && c != 0)? assert((a / b == c) && (a / c == b)): assert(a == 0);	

```
1 Node [0] -> Node [1];
2 Node [1] -> Node [2] label=True Branch;
3 Node [1] -> Node [3] label=False Branch;
4 Node [2] -> Node [3];
5 Node [3] -> Node [4];
6 Node [4] -> Node [5] label=True Branch;
7 Node [4] -> Node [6] label=False Branch;
8 Node [5] -> Node [4];
9 Node [6] -> Node [7];
10 Node [7] -> Node [8] label=True Branch;
11 Node [7] -> Node [9] label=False Branch;
12 Node [8] -> Node [7];
```

Listing 4.7. Data structure produced by CFG Generator

5-) SIF Rename: allows users to change the names of functions, variables and contracts by locating and replacing all occurences. The tool accepts three additional inputs: identifier type, old name and new name. The *before* function first parses the command line options for these additional inputs. As SIF traverses the AST, the *visit* function checks if the currently visited node matches the type and name of the identifier provided by the user. If a match is found, the old name gets replaced with the new one. The replacement is also performed within other nodes that reference the provided identifier. For instance, if a user wants to rename a particular contract, `SIF Rename` will rename the contract within the `contract definition` AST node. It will also rename references to the contract within other nodes such as `member access` and `using for directive` that may reference the given contract.

6-) Fault Seeder: instruments the code by injecting vulnerabilities. For each type of

vulnerability illustrated in Table 5.2, the `Fault Seeder` tool creates a new code block containing that vulnerability and injects it into the Solidity contract. SIF traverses the contract AST and inserts an extra AST node containing a single vulnerability into the original AST. Solidity code is then generated from the modified AST, and is referred to as a mutated contract. For example, to introduce integer underflow, `Fault Seeder` inserts the code snippet shown in Listing 4.8 into the original smart contract.

```
1 uint256 minuend = 20;
2 uint256 subtrahend = 250;
3 uint256 result = minuend - subtrahend;
```

Listing 4.8. Code snippet to introduce underflow by Fault Seeder

7-) Assertion Analyser: uses AST query and instrumentation capabilities to insert assertions in the contract. Inserting property checks using assert statements is helpful in verifying program correctness. This tool inserts pre- and post-conditions for arithmetic operations to help detect overflow, underflow and division by zero vulnerabilities. Within the *visit* function, the `Assertion Analyser` tool first gathers information on the node, and operators and operands if any within. For example, an expression node with arithmetic operations may be prone to an overflow/underflow error. The tool then inserts relevant assert statements, shown in Table 5.2, as extra nodes following the node under analysis. For division by zero vulnerability, SIF inserts a pre condition, before the node under analysis with a division operator, that asserts the divisor expression is greater than zero. Consider the function *uint2str*, shown in Listing 4.6, with a subtraction operation on line 10. The `Assertion Analyser` tool will insert an assert statement, shown in Listing 4.9, to check for unsigned integer underflow.

```
1 uint k = len - 1;
2 assert(len >= k && len >= 1);
```

Listing 4.9. Inserted guard for a subtraction operation by Assertion Analyser

4.3 Experiment

We evaluate feasibility, ease of use and extent of automation in using SIF and the seven tools built on it over 1838 unique Solidity contracts from the Ethereum network. Additionally, after SIF was released on Github in May 2019, we received various enquiries about using the framework from different parts of the globe. The interest

expressed in the framework was encouraging. The feedback helped us fix bugs and improve the usability of the framework and is discussed in Section 4.4.4.

4.3.1 Research Questions

We investigate the following questions in our experiment:

Q1. Solidity Language Support: *Does SIF support all constructs in the Solidity syntax and generate code accordingly?*

To answer this question, we first inspect contract code in our dataset and the official Solidity documentation to check the range of syntactic structures present in the dataset. We then use SIF, without any instrumentations, to generate Solidity contracts from ASTs and we compare if the generated code is the same as the original.

Q2. Correctness of AST Query and Instrumentation: *Are the tools built on top of SIF able to query and instrument the AST and produce correct output?*

We use a small user group to help answer this question. The user group comprised of ourselves (`User-0`), 2 PhD students familiar with Solidity language (`User-1` and `User-2`), and a second year undergraduate student (`User-3`) with basic knowledge of Solidity. We manually check the correctness of the query and instrumentation information over the different contracts.

Q3. Feasibility, Extent of Automation and Ease of Use: *How feasible is the framework for users to write their own tools? Are the tools fully automated and easy to use?*

We asked `User-1`, `User-2` and `User-3` to evaluate the framework and the 7 tools on the following aspects:

1. **Ease of implementing new tools** On a scale of 1 (very hard) to 5 (very easy), rate how easy it is to implement new tools using SIF. We asked the users to write a new tool, `Loop Count`, to report the number of loops in a Solidity contract, that utilises the AST query capability of the framework. We also asked the users to create a tool, `Make Signed`, that changes all unsigned integer (uint) types to signed integer (int), utilising both query and instrumentation capabilities.
2. **Extent of automation of the 7 tools** On a scale of 1 (Completely manual)

to 5 (fully automated), rate the extent to which the seven tools can be run automatically.

3. **Ease of use of the 7 tools** On the scale of 1 (Very difficult to use) to 5 (Very easy to use), rate the ease with which the 7 tools can be used.

4.4 Result

We discuss the experimental results in the context of the experiment questions presented earlier in this Section.

4.4.1 Q1. Solidity language support

The 1838 Solidity contracts in our dataset contain a wide variety of Solidity constructs. We manually checked against the Solidity documentation as well as the source code of the official Solidity compiler, and confirmed that our dataset covers all syntactic elements in Solidity. Our technique was able to analyse the AST and generate source code for all 1838 smart contracts from their ASTs automatically. The generated contracts were also compared by the 3 users to the original contracts, and no differences were reported. As discussed in related work chapter, the F* tool [12] does not support loop structures in Solidity. A significant fraction of contracts in our dataset contain loops and SIF is able to fully support loops. Self-destructs, throw statements and inline-assembly blocks are not supported by the Zeus verification tool [58]. Our framework fully supports the use of these constructs in Solidity contracts.

4.4.2 Q2. Correctness of AST query and instrumentation

The seven tools discussed in Section 4.2.4 utilise SIF capabilities for AST query and instrumentation. We evaluate correctness of these capabilities in the context of the seven tools.

To validate output of tools `Function Listing` and `CG Generator`, we split the dataset into two halves. With contracts in the first half of the dataset, two users ran the tools to collect function lists and call graphs. The other two users validated the output produced by the tools by inspecting the original contract. We, then, swapped the roles of the users for contracts in the second half. This was done to reduce biases, if any, and the monotony in checking the contracts. The users did not find any anomalies in the tools over the given dataset.

Control flow graphs produced by `CFG Generator` were manually inspected by `User-0` to check its correctness. CFG generator correctly generated CFGs for all contracts in the dataset. To evaluate `AST Diff`, we split the dataset into two halves again. `User-0` and `User-3` modified contracts in the first half by renaming identifiers, mutating operators and deleting code blocks for the first half. `User-1` and `User-2` ran the `AST Diff` on each set of original and modified contract (from first half), to check if all the modifications were reported correctly by the tool. The user roles were swapped for contracts in the second half of the dataset. AST Diff was found to work correctly by all users over all the contracts.

For the instrumentation capability, we used `SIF Rename` to locate and change names of identifiers within contracts in the dataset. We again, split the user reponsibilities so that two users selected and changed identifier names and two users checked the changes were reflected in the generated Solidity code. Different identifier types were selected. The users found SIF Rename worked correctly for the given dataset and requested changes.

Each of the four users ran `Fault Seeder`, selecting different vulnerability types to be seeded in each Solidity contract, and inspected the mutated contracts to check if the vulnerabilities were seeded as per choice. The mutated contracts were also run using the Solidity compiler, `solc`, to check for syntactic correctness. No issues were reported. The `Assertion Analyser` tool was run on all contracts, and the users individually inspected if the assertions were inserted correctly for the arithmetic operations. The instrumented contracts with assertions were checked through the `solc` compiler for syntactic correctness; no problems were found. The users were able to confirm the assertions were inserted by the tool correctly around arithmetic operations in all contracts.

4.4.3 Q3. Extent of automation and ease of use

Table 4.2 reports how `User-1`, `User-2`, and `User-3` rate the existing 7 tools for ease of use and extent of automation. The table also shows how the users rate feasibility and ease with which new tools, `Loop Count` and `Make Signed`, can be implemented using SIF. As the user interface for the existing 7 tools is similar, we report the average ratings across the 7 tools for ease of use and extent of automation for each user. `User-1, User-2, User-3` rate the extent of automation for the 7 tools as 4, 5 and 4, respectively, giving a high average user experience of 4.3 (mostly fully automated). This implies that

Table 4.2. User Experience of Using SIF and Tools

User #	Existing 7 Tools		Using SIF	
	Extent of Automation	Ease of Use	Ease of Implementing New Tools	
			Loop Count	Make Signed
1	4	4	4	4
2	5	4	5	4
3	4	5	4	3
Avg.	4.3	4.3	4.3	3.7

the users agree that the tools have a high level of automation with little user intervention. The users gave a score less than 5 as the users had to use the Solidity compiler to first produce the AST and then call the tool. We will look to automate this step in the future so the users do not have to explicitly call the Solidity compiler.

For ease of use of the 7 tools, all three users felt the interface for running the tools was easy to understand and use (ratings of $4, 4$ and 5).

For writing new tools with SIF, all three users felt it was easy to implement the `Loop Count` tool (average implementation ease being 4.3) owing to implementation similarity to the existing `Function Listing` tool (replace functions by loops). The users found implementing the `Make Signed` tool was slightly more challenging, as it is different from implementations of the existing 7 tools. Implementing the `Make Signed` tool requires searching through variable declarations, function parameter lists to locate all unsigned integer qualifiers and replacing them with the signed qualifier. `User-3` found implementation of the `visit` function was only slightly easy (rating of 3), while `User-1` and `User-2` found it easy (rating of 4). User-3 is not proficient in C++, and found the use of pointers in the *visit* function interface tricky. To address this issue, we have added comments in the interface showing how to use and cast to AST node pointers. Average ease of implementation across the users was 3.7 for the `Make Signed` tool.

Finally, we report the time taken in using the framework to generate Solidity code from unmodified ASTs. For contracts with less than 1000 lines of code (90.0% of the dataset), the framework finished code generation in 4 seconds. One of the bigger contracts, Expiry[3] with 6183 lines of code, takes 85.6 seconds for code generation from AST. To understand why code generation for this contract took significantly longer, we

[3]The contract Expiry is available at `https://etherscan.io/address/0x0ece224fbc24d40b446c6a94a142dc41fae76f2d`

profiled the framework phases and found that 40.3 seconds was spent on parsing the JSON format AST. Our framework uses the text format AST as the main reference and only queries the JSON format AST when it encounters keywords missing in the text AST. However, each such query starts from the root of the JSON AST for each newly visited node, resulting in significant overhead. Other contracts with more than 5K lines of code finished code generation in less than 15 seconds as they did not have to refer to the JSON AST very frequently. In our future work, we will optimise and accelerate queries involving JSON data structures.

4.4.4 Feedback from the Global Community

After SIF was released on GitHub, we noted interest in its use and received queries from users around the world. In this Section, we present feedback and suggestions from global users.

a) Bug report related to uninitialised variables: This was reported by a user from Ohio State University. We found the bug was caused by uninitialised smart contract variable definitions. The framework looks for the missing initialisation statement in such variable definitions. We have now fixed this bug.

b) Enquiry on gathering information from the AST: 2 users did not know how to maintain and summarise information gathered from different AST nodes. We provided them with guidance and examples of using *before* and *after* functions to address this issue. We also updated our documentation to include this.

c) Request for more Command Line options: A user made this request. After each SIF run, smart contract source code is generated by default. However, code generation is not always desired, especially for users who only want to query information from the AST. We have now provided this option to avoid code generation. We plan to add more command line options to SIF for additional user control.

d) Request for detailed documentation: A user requested more detailed documentation on each type of AST node and the associated fields and methods. We have provided this documentation for all types of AST nodes, along with examples using them.

4.5 Summary

SIF provides the capability to query, analyse and instrument the AST of Solidity contracts and generate Solidity code back. The framework uses a C++ intermediate

representation, and provides helper methods to gather information on different AST nodes and manipulate them, as needed. The framework is generic and eases the implementation of custom query and instrumentation capabilties. Users interact with the framework using a simple interface that is easy to understand and use. We built 7 tools for different types of Solidity code query and instrumentation to evaluate SIF's versatality and asked 3 users to rate the usability and automation of the tools along with ease of implementing new tools. We used 1838 unique contracts in our evaluation.

We found SIF was able to run all 1838 contracts, and the outputs of all 7 tools were confirmed to be correct by all the users. Additionally, the users found the 7 tools were ease to use and nearly fully automatic. The users also implemented 2 new tools using SIF and found the framework was well suited for implementing new tools, owing to the helper functions and simple interface. SIF's generality and ease of use makes it a promising framework for Solidity developers and testers to quickly and easily build, maintain new tools or leverage existing capabilities. Since the release of the framework on GitHub, we have received interest in its use for Solidity code analysis and instrumentation from several international users. We plan to add features including AST matchers to help users locate interested nodes in the AST more conveniently and support diagnostics, easy development and maintenance of Solidity tools.

Chapter 5

SolAnalyser: A Framework for Analysing and Testing Smart Contracts

5.1 Introduction

The research contributions of this chapter is the design and implementation of the our analysis and testing framework for Solidity smart contracts. This chapter describes the approach, the implementation and evaluation of the framework using 1838 real smart contracts from which we generate 12866 mutated contracts by artificially seeding 8 different vulnerability types.

The techniques we propose in this chapter will allow complete *automated analysis* of smart contracts, using both static and dynamic techniques to reduce the number of false positives, and handle the entire syntax of smart contracts. To help evaluate the rigor and effectiveness of different analysis tools, we developed a fault seeder that can inject well known vulnerabilities in smart contracts. The contributions in this chapter are as follows,

1. **Static checks**: Statically analyse source code of Solidity smart contracts to assess locations prone to vulnerabilities. We then instrument the source code with assertions that act as correctness property checks.

2. **Test generator**: We built an automated input generation tool for smart contracts, referred to as `InputGenerator`, that provides inputs for all transactions and functions in a contract.

3. **Runtime Monitoring**: We trigger the presence of vulnerabilities when the property checks are violated during execution of smart contracts on the Ethereum Virtual Machine (EVM). The smart contracts are executed with inputs provided by our `InputGenerator`. The tool chain that combines static and dynamic checks along with input generation (Contributions 1, 2 and 3) is referred to as `SolAnalyser`.

4. **Fault seeding tool**: To help evaluate the effectiveness of `SolAnalyser` and compare it with existing analysis tools, we create a smart contract mutation tool, *MuContract*, that takes the original contracts and creates several faulty versions based on common vulnerabilities observed in real contracts.

5. **Empirical Evaluation**: We evaluate the effectiveness of `SolAnalyser` in detecting vulnerabilities on 1838 real contracts and their faulty versions. We compare precision and recall achieved by our technique against five recent popular analysis tools: Oyente, Securify, Maian, SmartCheck and Mythril.

We found `SolAnalyser` with static and dynamic checks supported by test generation, was effective at detecting vulnerabilities across all 1838 contracts and the 12866 mutated versions, with a precision of 72% and recall rate of 100%. Our technique was capable of detecting more types of vulnerabilities than all five existing analysis tools used in our experiment. Securify performs well in detecting arithmetic vulnerabilities - overflow, underflow and division by zero but has limited support for other vulnerabilities. Oyente performed poorly with low precision(9%) and recall (42%). Maian does not provide adequate support for different vulnerabilities but is good at detecting unchecked send vulnerability. SmartCheck and Mythril support 4 to 6 vulnerability types but precision and recall rates were not high, with Mythril doing better than SmartCheck. Finally, `SolAnalyser` had a lower analysis overhead than all 5 existing tools, scaling easily to larger contract sizes. Overall, `SolAnalyser` for automated smart contract analysis outperforms existing analysis tools in terms of support, scalability, and accuracy.

The rest of this chapter organised as follows. Our approach for static and dynamic analysis, test generation and fault seeding is discussed in Section 5.2. Experiment setup and research questions of our evaluation is discussed in Section 5.3. Section 5.4 presents the experimental results in the context of the research questions. Finally, Section 5.5 concludes this chapter.

5.2 Approach

5.2.1 SolAnalyser Overview

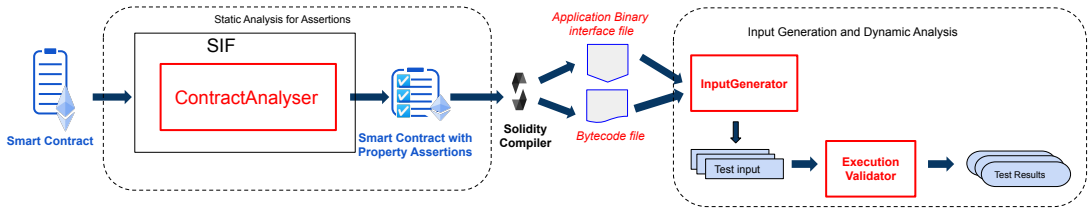

Figure 5.1. SolAnalyser: Assertion injection, input generation and analysis

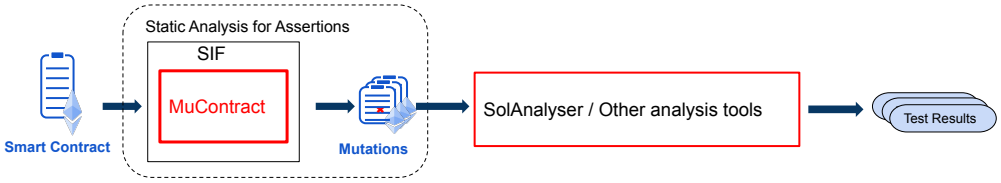

Figure 5.2. MuContract: Artifically seed vulnerabilities in a Solidity Contract to produce Mutated/Buggy Contracts.

Our approach for analysis of smart contracts includes three main components,

1. A vulnerability detection technique, `SolAnalyser`, shown in Fig. 5.1, that combines static analysis, implemented in `ContractAnalyser` using SIF [88] which is a code instrumentation framework for Solidity, and dynamic analysis, implemented in `ExecutionValidator`.

2. An automated input generation for smart contracts, implemented in `InputGenerator`

3. A tool, `MuContract`, that artifically seeds different types of vulnerabilities in smart contracts, using SIF. `MuContract` is used to assess effectiveness of `SolAnalyser` in revealing the seeded vulnerabilities. We also use the mutated contracts generated by `MuContract` to assess other existing analysis tools in the literature and compare with `SolAnalyser`, presented in Section 5.4.

We discuss each of these components in the following Sections.

5.2.2 Vulnerability Detection

Our approach for vulnerability detection has three phases:

1. Instrumentation with assertion via SIF.
2. Input generation for instrumented smart contracts.
3. Execution and analysis of instrumented contracts.

As seen in Fig. 5.1, we use ContractAnalyser to generate smart contracts with property assertions in phase 1. We automatically generate inputs for the smart contract in phase 2 using ABI and bytecode files. Phase 3 uses Ethereum virtual machine for executing the smart contracts accompanied by dynamic analysis of the execution traces implemented in ExecutionValidator. We describe each of the phases in the rest of this Section.

Phase 1: Instrumentation with assertion via SIF.

This phase implemented in `ContractAnalyser` performs instrumentation using Solidity Instrumentation Framework, SIF [88]. We first implement a *visitor* that traverses all the nodes in the smart contract AST. When a node is visited, the ContractAnalyser first checks whether the node type is susceptible to any of the well known vulnerabilities. It also checks the operator type and operand types within the node. For example, an expression node with arithmetic operations may be prone to an overflow/underflow error. The analyser then generates relevant assert statements on the result of the operations in the node that flags a vulnerability when the assert statement fails. The assert statement node is added after the node under analysis in the program control flow. For division by zero vulnerability, we insert a pre condition, before the node under analysis with a division operator, that asserts the divisor expression is greater than zero. Table 5.1 illustrates the different types of vulnerabilities checked by the ContractAnalyser and the corresponding assert and pre conditions inserted. Source code for our tools is available at `https://github.com/sefaakca/SolAnalyser`.

Phase 2: Input generation for smart contracts.

In this phase, we automatically generate inputs for instrumented smart contracts. First, we use Solidity compiler to generate Application binary interface (ABI) and bytecode files. ABI file holds information regarding functions in the contracts such as function name, function type, input types, and output types etc. Bytecode file holds the predefined bytecode of the smart contract. Then, using these two files, we generate inputs for smart contracts. Our input generator, implemented in Java, supports all Solidity types such as signed/unsigned integers types with widths ranging from 8 to 256. The generated inputs call each of the functions in the smart contract at least once. Source code for the input generator is available at `https://github.com/sefaakca/SolAnalyser`.

Table 5.1. Types of Vulnerabilities and Assertions for Property Check

Operation	Type of vulnerability	Assertions for property check
Addition a = b + c	Unsigned overflow	Post-condition: a >= b && a >= c
	Signed overflow/underflow	Post-condition: (c >= 0 && a >= b) || (c < 0 && a <b)
Subtraction a = b - c	Unsigned underflow	Post-condition: b >= a && b >= c
	Signed overflow/underflow	Post-condition: (c >= 0 && a <= b) || (c < 0 && a > b)
Multiplication a = b * c	Unsigned overflow	Post-condition: (b != 0 && c != 0)? (a >= b && a >= c) : (a == 0)
	Signed overflow/underflow	Post-condition: (b != 0 && c != 0)? (a / b == c && a / c == b) : (a == 0)
Division a = b / c	Division by zero	Pre-condition: c != 0

Phase 3: Execution and analysis of instrumented contracts.

The instrumented Solidity code with property assertions is executed in the Ethereum Virtual Machine (EVM) [52]. We use the inputs created in Phase 2 to execute the contract so that each transaction and function within the contract is invoked at least once. We implemented the *ExecutionValidator* in nodejs–v8.11.3–as an extension to EVM to analyse the execution trace produced by EVM and to report triggered vulnerabilities, if any. We use five different external node modules (`ethereumjs-vm`, `ethjs-signer`, `ethjs`, `crypto-js`, and `web3`) to combine EVM and execution trace analysis in the ExecutionValidator. ExecutionValidator reads the input file produced by the InputGenerator for a given contract. Each transaction field in the input file is sent to the EVM with a unique id, account address, input variables and bytecode of the contract for execution. Upon execution, EVM returns the execution trace in the form of runtime opcodes, similar to assembly code. ExecutionValidator checks the runtime opcodes for the presence of certain keywords (shown below) that indicate failure of instrumented assertions and consequently, presence of vulnerabilities. The keywords that ExecutionValidator scans for in the execution trace are,

- *TIMESTAMP*: timestamp usage
- *ORIGIN*: transaction origin usage
- *INVALID*: integer overflow, underflow or division by zero
- *CALL-REVERT*: unchecked send
- *CALL*: repetitive call function

If any of these keywords are present, an error showing the vulnerability type, function name in the code and the transaction id that triggered it is reported to the user.

For *out of gas* vulnerability, ExecutionValidator compares cost of transaction against the gas limit set at the time of execution. The gas limit is set before the transaction using Solidity compiler (*solc –gas*). *Solc –gas* command print an estimate of the maximal gas usage for each function. The cost of the transaction is computed by EVM using opcodes in the transaction and gas usage associated with each opcode. If the transaction cost exceeds the set limit, an error containing gas limit, transaction cost and the function invocation that triggered it is reported.

5.2.3 MuContract: Fault Seeding Tool

`MuContract` seen in Fig. 5.2, takes a Solidity contract as input and produces mutated contracts such that each of them have a single artificially seeded vulnerability. MuContract operates on the AST of a Solidity contract (generated using the solc compiler). To seed vulnerabilities, we use the AST representation and helper functions in SIF. We modify the AST to seed a particular vulnerability by creating a new AST node containing the vulnerability. Description of the vulnerabilities and the corresponding statements inserted is presented in Table 5.2. Each seeded vulnerability generates a separate modified AST from the original. Solidity code is then generated for each of the modified ASTs, referred to as mutated contracts. It is worth noting that only for the out of gas vulnerability in Table 5.2, mutation is not done at the AST level but rather by changing parameters of the execution environment in ExecutionValidator. Finally, we feed each of the mutated Solidity contracts to analysis tools such as SolAnalyser to evaluate their effectiveness in the detecting the seeded vulnerabilities. Source code of `MuCOntract` is available at `https://github.com/sefaakca/SolAnalyser`.

Table 5.2. Vulnerability Types Seeded Using MuContract

Division by zero	Statement with division by zero is inserted.
Out of gas	Gas limit changed in the exec. environment.
Overflow	Computation with overflow inserted.
Underflow	Computation with underflow inserted.
Timestamp dependency	Assignment expression with block timestamp.
TxOrigin	Condition stmt using the value of tx.origin.
Unchecked send	Send function without any pre or post condition is inserted.
Repetitive call	A loop with a Send function is inserted.

5.3 Experiment

We evaluate the feasibility and effectiveness of `SolAnalyser` in uncovering vulnerabilities in 1838 real smart contracts and their mutated versions. The mutated versions were generated by our tool, `MuContract`, by seeding one of 8 different vulnerability types (discussed in Section 2.2.2) into each of the original 1838 contracts. We compare effectiveness of `SolAnalyser` against popular analysis tools, Oyente, Securify, Maian, SmartCheck and Mythril. We chose these five tools based on: 1. Release date and

feasibility of running the tool. We picked tools from the last 2 years that were well maintained and documented. We also checked feasibility of installing and running them. For instance, ContractFuzzer [56], although recently released in 2018, was not feasible to install and run even after multiple communications with the authors, 2. Popularity of the tool, based on citations and number of users. We investigate the following questions in our evaluation of `SolAnalyser`:

5.3.1 Research Questions

Q1. Extent of vulnerabiltiy support: *What different vulnerability types are each of the analysis tools capable of detecting?*

To answer this question, SolAnalyser and the five existing tools were run on mutated contracts representing several instances of 8 different vulnerability types discussed in Section 2.2.2. We analysed the output files of the tools to assess whether they were capable of detecting each of the 8 different vulnerabilities.

Q2. Effectiveness of SolAnalyser: *What is the precision and recall achieved by SolAnalyser in revealing vulnerabilities in the mutated contracts?*

We use SolAnalyser over the mutated contracts to check if the assertions and opcode analysis reveal the seeded vulnerabilities. `InputGenerator` within `SolAnalyser` produces inputs from the original unchanged contract in JSON format. The `InputGenerator` tests each of the functions in the contract with 100 different input values.

Q3. Comparing Effectiveness of SolAnalyser with Oyente, Securify, Maian, SmartCheck and Mythril: *Which analysis tool is most effective in revealing different vulnerability types?*

We first run all six tools SolAnalyser, Oyente, Securify, Maian, SmartCheck and Mythril on the original source code of each of the 1838 smart contracts to assess their capability in analysing and revealing vulnerabilities. These smart contracts have been published and are being used, so we do not expect to find vulnerabilities in them. We then evaluate the effectiveness of the six tools in uncovering seeded vulnerabilities in the mutated contracts generated by `MuContract`. We report their effectiveness using precision and recall rate [49] that are widely used to evaluate performance of classification techniques. Precision, in our context, measures the proportion of actual vulnerabilities detected by the tool among all those that exist in the mutated contracts. Recall measures the proportion of actual vulnerabilities among all the vulnerabilities reported by the tool. Positive label is used for presence of vulnerability and negative

class for absence. Thus, True Positive (TP) refers to mutated contract reported by a tool as vulnerable with the correct location of seeded vulnerability. False Positive (FP) is when the tool does not report the seeded vulnerability in a mutated contract. False Negative (FN) occurs when the tool reports a vulnerability but at a location *different* from the seeded vulnerability in a mutated contract. Precision and Recall are computed as follows:

$$Precision = \frac{TP}{TP+FP}, \quad Recall = \frac{TP}{TP+FN} \tag{5.1}$$

5.3.2 Data Set

We collected 1838 unique and verified[1] smart contracts of different sizes from Etherscan [36]. Contract names and lines of code is available at `https://github.com/sefaakca/SolAnalyser`. The largest contract in our experiment has 6183 LOC and average LOC across contracts is 469. From each verified smart contract, we create mutated contracts by seeding each of eight different vulnerabity types. In total, we generate 12866 mutated contracts for our data set.

5.3.3 Oyente, Securify, Maian, SmartCheck and Mythril

We use the latest version of Oyente, Securify, Maian and Mythril provided in GitHub [35, 70, 73, 79] to analyse the 1838 contracts and 12866 mutated contracts. For analysis with SmartCheck, we load the smart contract into the web IDE [102] and use the analyse feature. All six tools in our experiment use Solidity compiler versions, 0.4.25 and 0.5.3, based on the syntax of the contract.

5.4 Result

In this Section, we report and discuss results in the context of the research questions presented in Section 5.3.

5.4.1 Q1. Extent of Vulnerability Detection

Running `SolAnalyser` and the existing 5 tools on 12866 mutated contracts revealed the vulnerability types supported by each tool, shown in Table 5.3. We find SolAnalyser

[1] source code and byte code conform with each other

Table 5.3. Extent of Vulnerability Support

Name of the tool	divbyzero	overflow	underflow	timestamp	tx.origin	uncheckedsend	reccall	outofGas
SolAnalyser	✓	✓	✓	✓	✓	✓	✓	✓
Oyente	✗	✓	✓	✓	✗	✗	✗	✗
Securify	✓	✓	✓	✓	✗	✗	✓	✗
Maian	✗	✗	✗	✗	✗	✓	✗	✗
SmartCheck	✓	✓	✓	✓	✓	✓	✓	✗
Mythril	✗	✓	✓	✗	✓	✓	✗	✗

has the widest support for vulnerability types, handling all 8 types of vulnerabilities. SmartCheck support 7 of the 8 types of vulnerabilities. Securify and Mythril have reasonable support, handling 4 of the 8 vulnerability types. Maian performs worst in this aspect, only being able to detect uncheckedsend, with no support for the other 7 vulnerability types. With regards to vulnerability types, integer overflow and underflow are detected by 5 of the 6 tools, except Maian. Outofgas is the vulnerability with weakest tool support. `SolAnalyser` is the only tool in our experiment that can detect this vulnerability.

5.4.2 Q2. Effectiveness of SolAnalyser

In this Section, we evaluate the effectiveness of SolAnalyser using the mutated contracts, reporting precision and recall for different vulnerability types. Precision and recall rate of the SolAnalyser over all the mutated contracts is shown in Fig. 5.3 and 5.4, respectively. Comparison with other tools shown in the figures is discussed in the next Section 5.4.3. Recall rate of SolAnalyser for each vulnerability type is 100%. This implies SolAnalyser does not generate any false negatives, i.e., it does not erroneously report vulnerabilities at locations that are mutation free. Precision rate of SolAnalyser for out of gas vulnerabilities is 100%, implying that all the seeded out of gas vulnerabilities were revealed by our tool. Precision for timestamp dependency was 75% and the remaining vulnerabilities between 57 – 60%, implying that some of these vulnerabilities were not caught by our tool. The reason for this was because the vulnerabilities were embedded within multiple conditional statements (with conditions, for example, checking type of user and account balance) that were not reached by the inputs from `InputGenerator`. For a sample of 150 contracts, we manually set the inputs to reach these conditions, which allowed `SolAnalyser` to then reveal the seeded vulnerabilities. Enhancing the `InputGenerator` component in our tool to achieve complete control flow coverage within transactions and functions in the contract will help increase the

precision in detecting different vulnerabilities.

5.4.3 Q3. Comparing Effectiveness of SolAnalyser with existing tools:

In this Section, we compare the precision and recall achieved by `SolAnalyser` for the different vulnerability types against 5 existing tools: Oyente, Securify, Maian, Smartcheck, and Mythril. With respect to precision, `SolAnalyser` achieves superior performance in detecting out of gas, timestamp dependency, tx.origin, and unchecked send over existing tools. Recall rate for SolAnalyser is at 100% for all vulnerability types, significantly better than existing tools. Securify has the best precision in detecting integer overflow, underflow, division by zero and repetitive call. SolAnalyser has comparable precision to Securify in detecting repetitive calls (56%), and 10% less precision than Securify in detecting overflow, underflow and division by zero as the inputs used by `SolAnalyser` do not reach the vulnerability in some instances. Nevertheless `SolAnalyser` outperforms Securify in recall rate for integer overflow, underflow, division by zero, and repetitive calls. SmartCheck also reports overflow, underflow and division by zero vulnerabilities but with low precision and recall. The patterns specified in SmartCheck to detect these vulnerabilities are not complete and disregards checks inserted by the developer. Mythril has low precision (< 0.4) and reasonable recall (0.65) in detecting overflow and underflow. Mythril performs boundary checks, assuming uint256 type, in arithmetic operations to detect these vulnerabilities. For signed and unsigned integer widths that are different from uint256 (Solidity supports signed and unsigned integer widths from 8 to 256 bits), Mythril is prone to reporting false positives and false negatives. Oyente supports detection of integer overflow and underflow has very poor precision (< 0.1) and recall (< 0.3) as it disregards developer provided checks for overflow, underflow and cannot handle input numbers that exceed the boundary.

Maian only supports detection of unchecked send vulnerabilities with low precision (0.19) and high recall (0.95). Transaction origin usage vulnerability is best detected by `SolAnalyser` but is also supported by Mythril and Smartcheck, albeit with low precision as they miss tx.origin keywords embedded in loops or conditions. Out of gas vulnerability is only supported by `SolAnalyser` with 100% precision and recall. Identifying out of gas vulnerability requires access to parameters in the runtime environment, the capability for which is not available in other tools. ExecutionValidator provides this

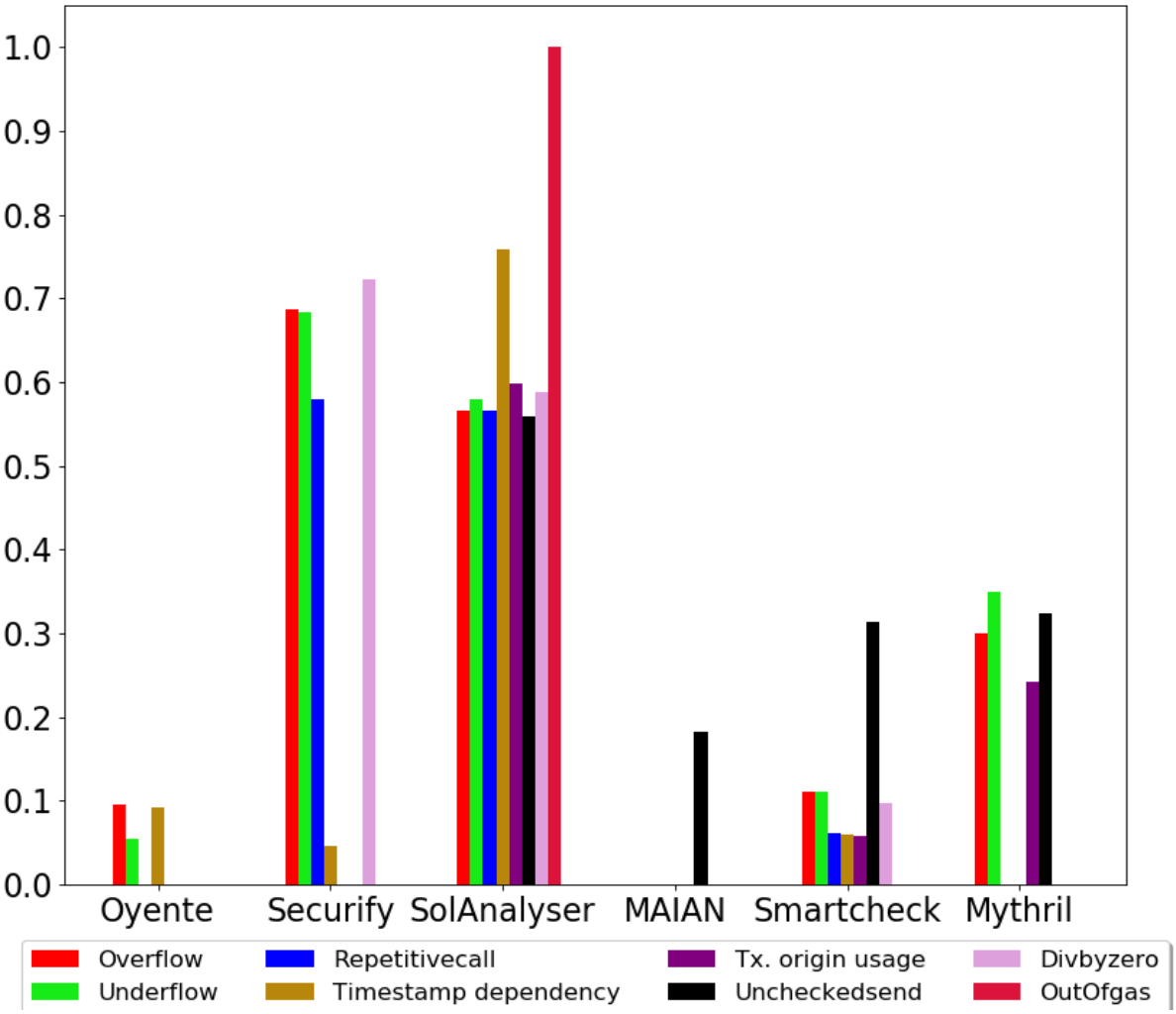

Figure 5.3. Precision rate of Oyente, Securify, SolAnalyser, Maian, SmartCheck and Mythril for mutations

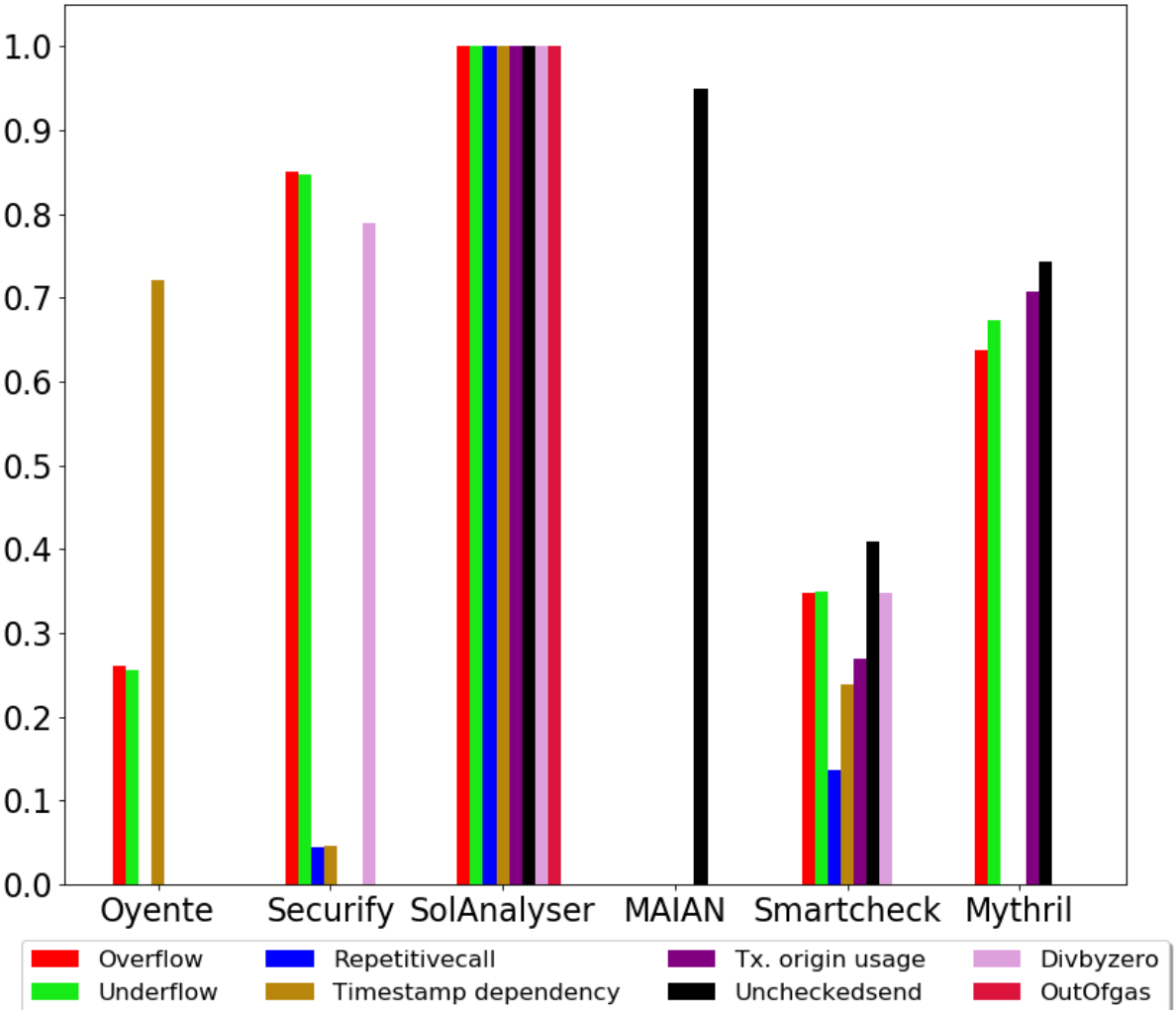

Figure 5.4. Recall rate of Oyente, Securify, SolAnalyser, Maian, SmartCheck and Mythril for mutations

access within `SolAnalyser`.

In summary, across all vulnerability types and mutated contracts, we find `SolAnalyser` easily outperforms existing tools taking precision and recall into account. Precision of `SolAnalyser` is lower than Securify for arithmetic vulnerabilities - overflow, underflow and division by zero. Precision of `SolAnalyser` for these vulnerabilities can be improved with better inputs from InputGenerator. Recall of `SolAnalyser` is, however, better revealing that it does not erroneously report vulnerabilities. For the other 5 vulnerability types, `SolAnalyser` achieves both better precision and recall than all 5 existing tools.

Analysis time taken by SolAnalyser versus existing tools. For analysis using the same hardware, we found average time taken by each tool per contract to be as follows in ascending order, 1. `SolAnalyser` = 13.5 secs, 2. Maian = 20.1 secs, 3. SmartCheck = 23.5 secs, 4. Oyente = 26.9 secs, 5. Securify = 67.2 secs, Mythril = 167.9 secs. We find `SolAnalyser` takes the least amount of analysis time, and scales easily to large contract sizes.

5.5 Summary

We have proposed a fully automated technique for vulnerability detection in smart contracts that uses code instrumentation and execution trace analysis. We present detection of 8 vulnerability types that have limited analysis support in literature. Our framework for inserting property checks in Solidity code is generic and provides interfaces that can easily be used to support detection of other types of vulnerabilities. We perform dynamic analysis on the Ethereum virtual machine. We also provide support for aritificially seeding different types of vulnerabilities in Solidity contracts. The mutated contracts can be used to assess effectiveness of analysis tools in revealing the seeded vulnerabilities.

We used 1838 verified contracts in our evaluation and generated 12866 mutated contracts by artifically seeding each of 8 different vulnerability types. We used the mutated contracts to analyse precision and recall of `SolAnalyser` and five other popular existing tools - Oyente, Securify, Maian, SmartCheck and Mythril. We find `SolAnalyser` has the widest support for vulnerabilities, supporting all 8 types unlike existing tools. Precision (average of 72%) and Recall (average of 100%) is significantly better than existing tools across all vulnerability types in the mutated contracts. We find precision of `SolAnalyser` can be further enhanced by improving the quality of inputs

produced by the InputGenerator component. Finally, analysis time overhead is lowest for `SolAnalyser` (13.5 secs) compared to existing tools.

Chapter 6

Testing Smart Contracts: Which Technique Performs Best?

6.1 Introduction

The research contributions of this chapter is conduct an empirical evaluation of testing techniques for smart contracts. The testing techniques we evaluated are: (1) Blackbox fuzzing, (2) Adaptive fuzzing, (3) Coverage-guided fuzzing with an SMT solver and (4) Genetic algorithm. This chapter describes the implementation of these four testing techniques and evaluation effectiveness of the test generation techniques using (1) Coverage achieved - we use four code coverage metrics targeting smart contracts, (2) Fault finding ability - using artificially seeded and real security vulnerabilities of different types.

Several smart contract test input generation techniques have been proposed in the literature to detect security vulnerabilities [3, 48, 56, 81]. Many of the existing techniques use fuzzing approaches, namely (1) Blackbox Fuzzing (BF) - that relies on random input generation based on contract interface [3, 48, 56], and (2) Adaptive Fuzzing (AF) - feedback-guided input generation to generate high quality inputs that combines genetic algorithms with fuzzing [81]. Coverage-guided fuzzing and genetic algorithms have been used effectively in other domains to generate high quality inputs [1, 38, 39, 68, 84, 93]. Consequently, we decided to support these additional approaches for testing smart contracts. We implemented the following techniques in this paper - (3) Coverage-Guided Fuzzing with the aid of an SMT solver (GF), (4) Genetic algorithm (GA) using opcode coverage as an objective.

We aim to compare the above four input generation approaches for Solidity smart

contracts with respect to their effectiveness in achieving code coverage, fault finding and overhead incurred. All the input generation approaches mentioned, except BF, are associated with a single implementation for Solidity - sFuzz [81] for AF, our native implementations for each of GF and GA.

We evaluated the four input generation techniques using two datasets of Solidity contracts – (1) Random-C comprising 1665 Solidity contracts with no known vulnerabilities, (2) Vulnerable-C containing 90 Solidity contracts with known vulnerabilities. We generated inputs with each of the techniques for a fixed time of 15 seconds per contract. We measured four Solidity code coverage metrics - branch coverage, opcode coverage, call coverage, event coverage achieved by the inputs over contracts in both datasets. We also seeded artificial faults for different types of security vulnerabilities in the Random-C dataset. We then assessed the effectiveness of the input generation techniques in uncovering the seeded faults in Random-C contracts. To mitigate the problem of artificial faults not being representative of real faults, we additionally performed fault finding on the Vulnerable-C dataset of 90 real contracts with known vulnerabilities gathered by authors in [32]. Among the four input generation approaches, we found AF was most effective at contract coverage and fault finding over both the Random-C and Vulnerable-C datasets. AF also generated the smallest input set with least overhead in execution.

The rest of this chapter organised as follows. Section 6.2 presents implementation of the testing techniques, definition and measurement of Solidity code coverage metrics. Section 6.3 presents experiment setup and research questions. Section 6.4 presents the experimental results in the context of the research questions. Section 6.5 discusses the threats to validity in our experiment. Finally, Section 6.6 concludes this chapter.

6.2 Approach

In this Section, we discuss the following, (1) implementations of the four Solidity testing techniques, (2) definition and measurement of four Solidity code coverage metrics, and finally, a (3) fault seeding framework for different types of vulnerabilities. We support Solidity compiler versions - 0.4.25 and 0.5.8 to compile the contract code in our experiment for Ethereum Virtual Machine (EVM) version 3.0.0. Our implementations for testing techniques, code coverage and fault seeding are publicly available at `https://github.com/sefaakca/TestingSCsWhichTechPerformsBest`.

6.2.1 Testing Techniques

6.2.1.1 BF

Instead of implementing new BF tool, we use our existing BF tool, SolAnalyser [3], in our experiments. BF implementation in SolAnalyser framework takes two arguments as inputs – the Application Binary Interface (ABI), and a bytecode file. These are generated from the Solidity contract with the Solidity compiler. ABI file holds information regarding functions in the contracts such as function name, function type, input types, and output types. Bytecode file holds the predefined bytecode of the smart contract. SolAnalyser implementation supports all Solidity types such as signed/unsigned integers types with widths ranging from 8 to 256 bits. The generated inputs call each of the functions in the smart contract and are written into a JSON file.

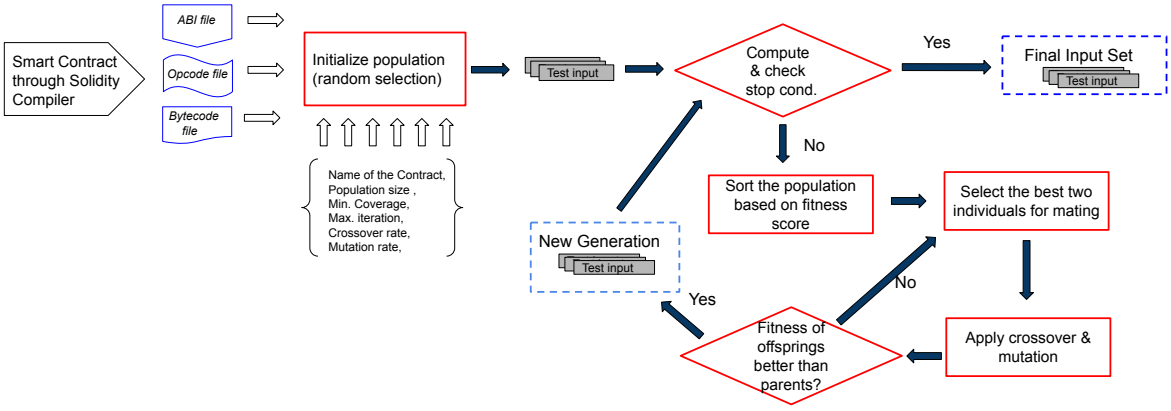

Figure 6.1. Genetic Algorithm working scheme

6.2.1.2 GA

Figure 6.1 shows the working scheme of the GA technique we implemented. We use Java for our implementation and take nine arguments as inputs. Four out of the nine input arguments are ABI, name of the contract, bytecode and opcode files. Opcode file holds the predefined opcode of the smart contract. We use these four arguments to generate the first random population. The remaining five arguments are directly related to GA operators – namely, size of the population, minimum desired coverage, number of iterations, mutation rate and crossover rate. Stopping condition for evolution is a pre-defined time of 15 seconds in our experiment. We use single-point crossover, along with user-defined crossover and mutation rate in our implementation. Mutation and crossover rates were set at 0.2. The crossover operator creates two offspring test inputs,

O1 and O2 from parent test inputs, P1 and P2. A value between 0 to 1 is chosen as the crossover rate, α. The first offspring contains α $\times|P1|$ test inputs from P1 and (1 - α) $\times|P2|$ test inputs from P2. The second one contains α $\times|P2|$ test inputs from P2 and (1 - α) $\times|P1|$ test inputs from P1. After the offsprings are generated, they are mutated by randomly updating some of the input values based on a user-defined mutation rate (between 0 and 1) that defines the extent to which the offspring is changed. Resulting offsprings are added to the new generation if they improve the fitness score achieved by their parents. We used opcode coverage as the fitness score. We measure opcode coverage of a candidate test input with our coverage measurement framework discussed in Section 6.2.2. We sort the population based on opcode coverage using fast-sorting dominance algorithm [27] with time complexity $O(MN^2)$, where N is the population and M is the number of objectives. After sorting, we select the top two test inputs to be parents for mating. We then apply crossover and mutation to generate offsprings. If any of the offspring test inputs achieve better fitness score than their parent test inputs, we add them to the population. We maintain population size by removing an equal number of poorly performing test inputs. We repeat this process until the stopping condition is met.

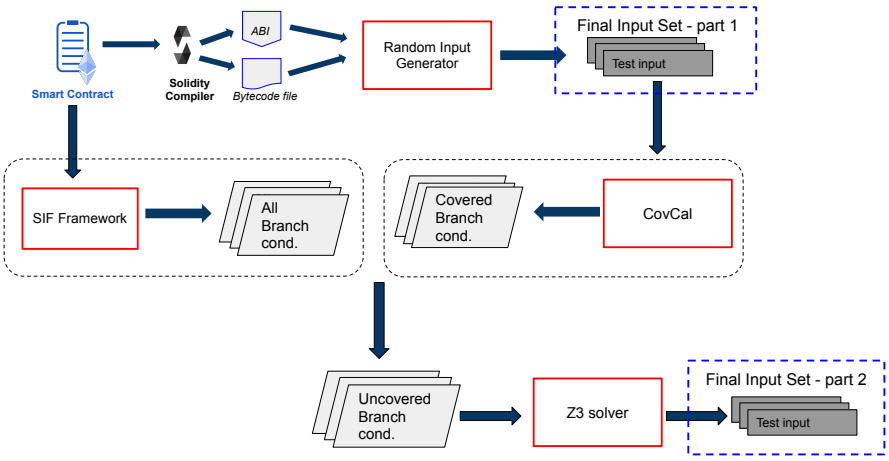

Figure 6.2. Coverage-Guided fuzzing with Z3 constraint solver

6.2.1.3 GF with Z3 constraint solver

Figure 6.2 shows the workflow of our GF test generation technique. We first generate random inputs for a fixed time (half the input generation time in our experiments) and measure branch coverage achieved using the coverage measurement tool, presented in Section 6.2.2. For the remaining half of the time, we gather path constraints for

uncovered branch conditions and feed them to the Z3 SMT solver. The solver returns inputs for uncovered branches that can be triggered. A Python script is then used to analyse uncovered branches and synthesise a Z3 program containing constraints to cover those branches. Finally, the Z3 program produces inputs for covering feasible conditions. The initial random inputs along with inputs from Z3 form the final input set.

Consider the following example contract named *Magi*.

```
function setValue(uint idx, uint newValue){
    if (idx == 0) {
        do something...
    } else if (idx == 1) {
        do something...
    } else if (idx == 2) {
        do something...
    } else if (idx == 3) {
        do something...
    } else {
        revert();
    }
}
```

Listing 6.1. setValue function from the Magi smart contract

With BF, we sample inputs from the range of a 256 bit unsigned integer. Majority of the random inputs execute the last `else` branch, since the likelihood of choosing values 0, 1, 2 and 3 in this range is low. GF technique, on the other hand, generates constraints to be solved for each condition checking *idx* variable values. The example below shows the contraint gathered by the instrumentation when visiting the function AST for the first if condition checking $idx == 0$.

```
#Initiate the Z3 solver
S = solver()
#Declare idx variable in Z3
#The UInt type is a customised Z3 type
idx = UInt('idx')
#Add the contraint to the solver
s.add(idx == 0)
#Solve the constraint
s.check()
```

Listing 6.2. Constraints generated

For all the conditions checking `idx` values, a unique Python script, resembling the above listing, is generated with different constraints on the values for `idx` corresponding to respective conditions. These constraints are then fed to the Z3 solver which in turn generates inputs for feasible conditions.

6.2.1.4 AF

We used the sFuzz tool implemented by Nguyen et al [81] as AF in our experiments. The source code of the tool is publicly available at `https://github.com/duytai/sFuzz`. We set the time for input generation in the sFuzz implementation to 15 seconds, matching the time set in the other tools.

6.2.2 Solidity Code Coverage

In this Section, we discuss the coverage measurement framework, `CovCal`, and the code coverage metrics we use in our measurement.

6.2.2.1 CovCal implementation

Executing the Solidity contract with an input in the EVM produces an execution trace. We implemented `CovCal` in nodejs v8.11.3 as an extension to EVM to analyse the execution trace produced by EVM. We capture both the execution result and progression steps returned by EVM. The execution result shows the return value of the function, validation code - 0 or 1, gas usage, gas refund, among other information. The progression step shows the execution trace in the form of runtime opcodes, similar to assembly code. The recorded execution result is used to determine gas usage. The recorded progression steps are used for calculating coverage metrics defined below,

6.2.2.2 Code Coverage Metrics

1) Opcode Coverage. Defined as fraction of opcodes executed by test inputs to total number of opcodes in the contract. To measure this, we record the opcodes executed at runtime. We removed duplicate opcode sequences that may occur within loops or repeated function calls. Total number of opcodes in the contract is inferred using the Solidity compiler. We calculate the opcode coverage achieved by a test input or input set using the following formula,

$$Opcode\ Coverage\ = \frac{\#Opcodes\ executed}{Total\ \#opcodes\ in\ contract} \tag{6.1}$$

`Brownie` [17] and `solidity-coverage` [98] use the same definition for Opcode coverage.

2) Branch Coverage. Measured as number of branches covered by the input set to total number of branches in the contract. We first compile a list of all the branches in the contract. Every branch executed by a test input in the input set is marked as covered. We then calculate branch coverage achieved by an input set using the following formula,

$$Branch\ Coverage\ = \frac{\#Covered\ branches}{Total\ \#branches\ in\ contract} \tag{6.2}$$

Branch coverage measured by `Brownie` [17] matches this definition.

3) Event Coverage. Measured as number of executed events to total number of events in the contract. For event coverage calculation, we first record opcodes executed by all the test inputs in the input set. From these recorded opcodes, we retrieve 'LOG' opcodes, since EVM will return this opcode whenever an event is triggered [116]. We calculate event coverage achieved by an input set using the formula below.

$$Event\ Coverage\ = \frac{\#Executed\ events}{Total\ \#events\ in\ contract} \tag{6.3}$$

4) Call Coverage. Defined as number of calls executed by test inputs in the input set to the total number of calls present in the contract. We record runtime opcodes executed and gather the ones that correspond to 'CALL'. Etherum yellow paper [116] confirms EVM will return this opcode whenever a call is triggered. We calculate call coverage using the formula below.

$$Call\ Coverage\ = \frac{\#Executed\ calls}{Total\ \#calls\ in\ contract} \tag{6.4}$$

We are not aware of any other existing tool supporting event or call coverage for Solidity code.

6.2.3 Fault Seeding

Fault seeding is also referred to as mutant generation where a mutant is a faulty contract with a single seeded fault. We use two existing tools for seeding faults in Solidity contracts - `MuContract` [3] and `Mutec` [87]. `MuContract` exists as part of the

SolAnalyser framework [3] to seed Solidity-specific vulnerabilities. `Mutec` is a tool for seeding operator faults in C/C++ programs [87]. We extend `Mutec` to support operator mutations in Solidity contracts as `MuContract` does not support this mutation type.

We generate ten mutations of different types for each Solidity contract. Seven out of the ten mutation types are specific to Solidity syntax. We use the `MuContract` tool to generate these mutation types, namely – integer overflow, integer underflow, division by zero, timestamp usage, tx.origin usage, unchecked send, repetitive function call [3]. The remaining three mutation types were traditional operator-related mutations for relational, logical and arithmetic operators. For these, we modified the `Mutec` tool to support Solidity syntax. `Mutec` is based on the Clang LibTooling framework that recursively visits the AST of C and C++ programs to seed faults of a given type. To support Solidity, we combine SIF [88], a Solidity instrumentation framework, with `Mutec`. `Mutec` marks locations of relevant operators for arithmetic, logical and relational mutations while SIF traverses the AST. Once SIF finishes visiting the entire AST, `Mutec` generates mutations for relevant operators in Solidity contracts.

The ten mutation types used in our fault seeder are inspired from known attacks on Ethereum smart contracts listed in [80]. We also support reentrancy mutations in our fault seeder implementation. However, our execution environment does not provide multiple threads to help simulate this attack. We, therefore, do not report on reentrancy mutations in our experiment.

For a given contract, fault finding ability of an input set is measured as

$$Fault\ Finding\ Score\ =\ \frac{\#Vulnerabilities\ Killed}{Total\ \#Vulnerabilities} \tag{6.5}$$

6.2.4 Execution environment

Execution environment helps determine the fault finding score for mutations and real vulnerabilities. Our execution environment is implemented as an extension to EVM to analyze the execution traces and report triggered vulnerabilities. We used external `node.js` modules to combine EVM and execution trace analysis. Execution traces from EVM are in the form of opcode sequences. To determine if a mutation is killed, we first run the original and mutated contract in the execution environment. We then compared their execution results, which comprise function return values, validation code, gas usage, and refund. Comparing execution results will suffice for certain mutation types like integer overflow, integer underflow, division by zero, arithmetic, logical, relational. For vulnerable contracts where the original correct contract is unavailable, we monitor

the execution and compute expected results using an external `big-integer` module in our execution environment that provides a larger value range to help avoid overflow or underflow in arithmetic operations.

For other mutation types, such as timestamp-usage, tx. origin, unchecked send and repetitive call, execution results between original and mutated contracts remain unchanged. For these types, we analyze the execution traces (opcode sequences) to detect mutations or vulnerabilities. For example, If we find a `Call` operation in an execution trace with no matching `Revert`, we signal the presence of an unchecked send vulnerability. If the vulnerability detected matches the description of a seeded fault or known vulnerability, we mark it as killed/detected.

6.3 Experiment

We evaluate effectiveness of the four testing techniques using two data sets - (1) *Random-C* - 1665 randomly sampled contracts from Etherscan that are assumed correct with no known vulnerabilities, and (2) *Vulnerable-C* - 90 contracts with known vulnerabilities from the SB curated dataset [101]. We fix input generation time at 15 seconds per contract for each of the four testing techniques. The time we used is comparable to input generation times in other contract testing papers [3, 56].

6.3.1 Research Questions

We investigate the following research questions:

Q1. Code Coverage comparison: *Which input generation technique achieves best code coverage on average?*

To answer this question, we generated input sets using BF, GF, AF and GA for the contracts in the Random-C and Vulnerable-C datasets. We then ran the generated input sets and measured branch coverage, opcode coverage, event coverage and call coverage using the `CovCal` framework.

Q2. Fault finding comparison: *Which test generation technique is most effective at fault finding on average?*

To answer this question, we assess fault finding using Random-C and Vulnerable-C datasets. Since contracts in Random-C have no known vulnerabilities, we seed 10 artificial faults (or mutants) in each of the 1665 contracts by analysing the source code and applying mutation operators to eligible locations using MuContract and Mutec.

Fault finding for a input set is measured as fraction of mutants killed (for seeded faults) or vulnerabilties detected (for real faults). We compare average mutation score achieved by the techniques across all 1665 contracts in the Random-C dataset and average fault finding across all 90 contracts in the Vulnerable-C dataset. Contracts in the Vulnerable-C dataset have been manually annotated (by the dataset creators) with the type of vulnerability and its location in the contract. When a test execution in our environment encounters a vulnerability, information on its location and type appears in the error message. We match this to the annotation in the dataset to mark the vulnerability as detected.

Q3. Input set size and execution time: *What is the size of the input set generated by each technique and how long does it take to run?*

We measured size of the input sets generated by BF, GF, AF and GA for each Solidity contract in the Random-C and Vulnerable-C datasets. We ran the test inputs on EVM to measure average execution time over 10 repeated runs of each input set. We used the same machine (Intel Quad Core i5-5200 CPU 2.20GHz, 8GB DDR3, 64-bits) for all runs. We report average input set sizes and average input set execution times across all contracts in the Random-C and Vulnerable-C datasets.

6.3.2 Data Set

Random-C: We collected 1665 verified[1] smart contracts of different sizes from Etherscan [36], ranging from 49 to 2856 LOC. We implemented a web-crawler to collect smart contracts whose source codes are publicly available on Etherscan web-site. We restricted the Random-C dataset to 1665 contracts in the interest of experiment resources and time as each of these contracts had a further 10 mutated contracts for the fault finding experiment. The contracts in our dataset covered a wide range of Solidity language features - 77 % of the contracts contain branch statements, 91% contain event usages, 57% comprise external calls, and finally, 27% contain loops.

Vulnerable-C: Durieux et al. [32] created the SB Curated dataset that contains Solidity contracts deployed in the Ethereum network with known vulnerabilities. Contracts in the dataset have been manually tagged with the location and type of its vulnerability. We used 90 vulnerable contracts with 5 different vulnerability types, namely- tx.origin usage (18) , integer overflow (7), integer underflow (8), timestamp usage (5), unchecked send (52).

[1] Source code and byte code of the contract conform with each other.

Dataset Availability: All the contracts in our datasets, implementation of testing techniques, coverage measurement and fault seeding, along with scripts to run the experiments can be found at `https://github.com/sefaakca/TestingSCsWhichTechPerformsBest`.

6.4 Result

In this Section, we report and discuss results in the context of the research questions presented in Section 6.3.

6.4.1 Q1. Code Coverage Comparison

We present opcode, call, event and branch coverage achieved by BF, GF, AF and GA input sets over the 1665 Solidity contracts in the Random-C and 90 Solidity contracts in the Vulnerable-C datasets in Figures 6.3 and 6.4, respectively. We discuss performance of the techniques with respect to each of the coverage metrics in the Sections below. We check if differences between the techniques is significant using one-way Anova and Tukey's honest significant difference (HSD) test. P-values for pairwise comparison of techniques using this test is shown in Tables 6.2 and 6.3 for the Random-C and Vulnerable-C datasets, respectively, with emphasis on values that show significant difference (at 5% significance level).

Uncovered Code. Median coverage achieved by all four input generation techniques is not that high (ranging from 31% to 78%) for contracts in both datasets in Figures 6.3 and 6.4. For instance, opcode coverage in Figures 6.3(a) and 6.4(a) show that more than 50% of the opcodes on average remain uncovered by all techniques. The main reasons for the large proportion of uncovered regions observed with all techniques are,
1. Restricted input generation time - In our experiment, we restrict input generation time with each technique to 15 seconds. Increasing the time for input generation will result in better coverage as seen in Table 6.1 which shows the average opcode coverage (over a smaller sample of 150 contracts) achieved by each technique with different generation times.
2. Dependency between functions not considered - All the techniques generate inputs to execute functions within a contract without considering dependency on other functions. As a result, conditions dependent on values from other functions may not be be satisfied. Statements depending on these conditions remain uncovered. An example is a smart

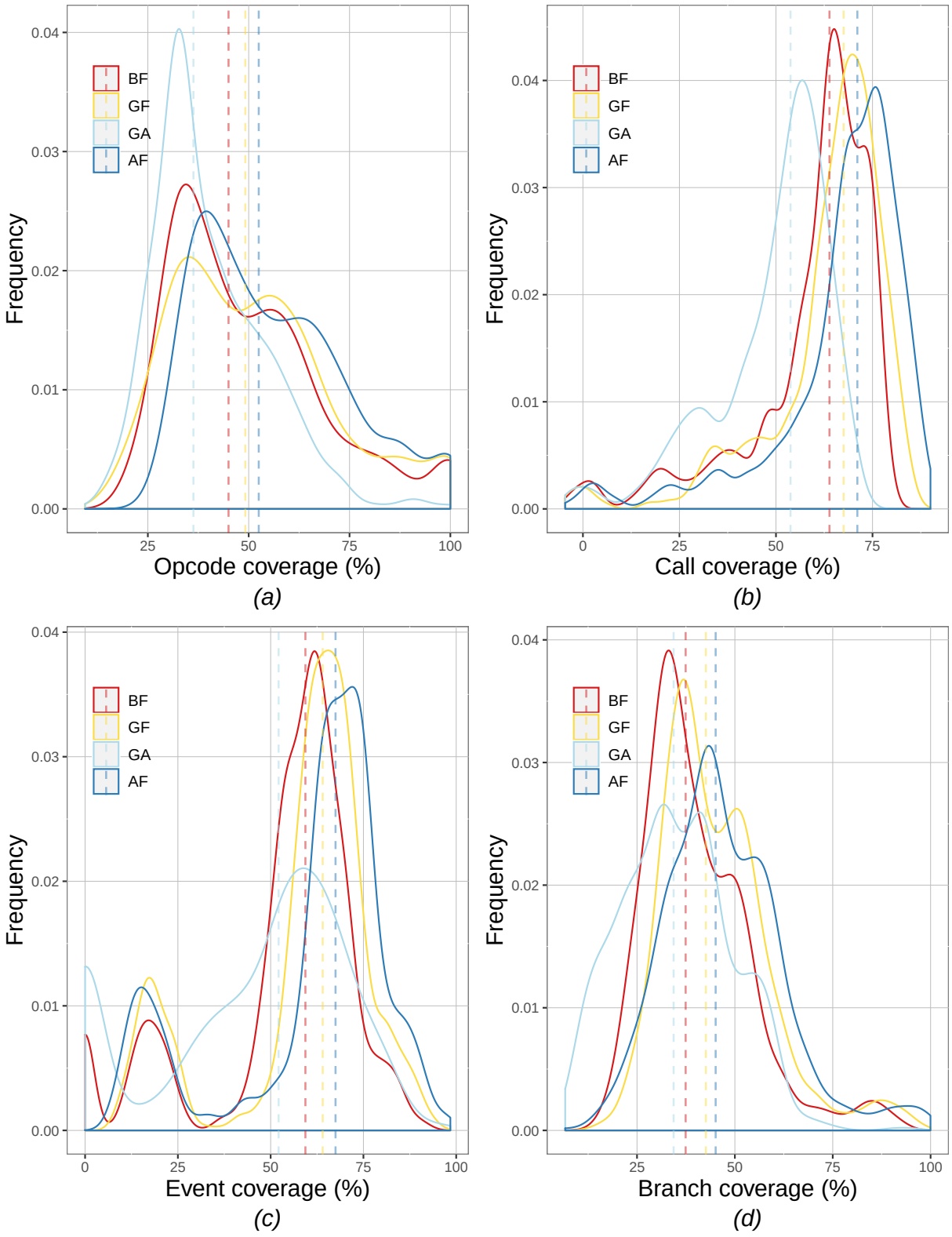

Figure 6.3. Histogram frequencies of coverage achieved by BF, GF, AF and GA input sets for 1665 Solidity contracts in the Random-C dataset

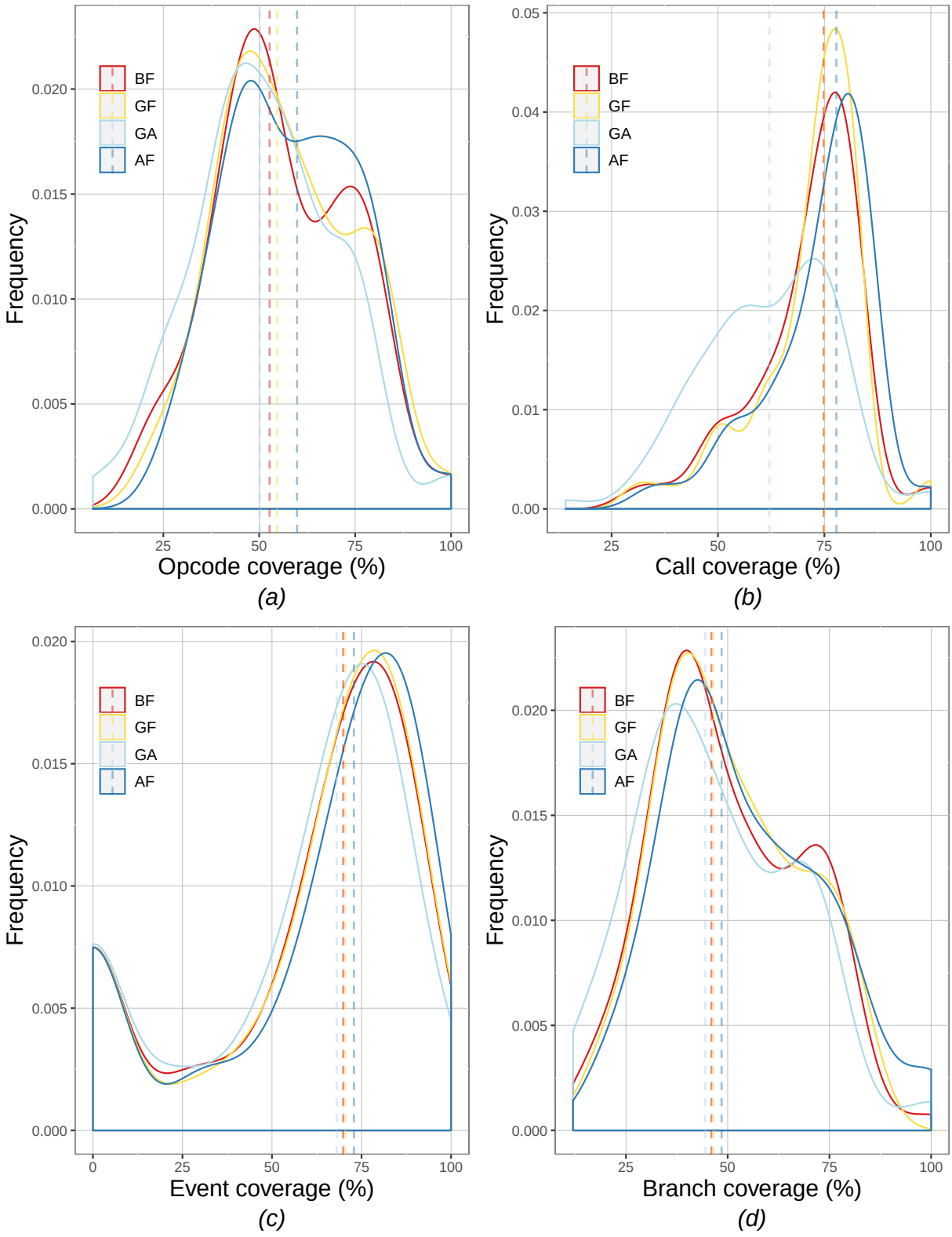

Figure 6.4. Histogram frequencies of coverage achieved by BF, GF, AF and GA input sets for 90 Solidity contracts in the Vulnerable-C dataset

Table 6.1. Average opcode coverage achieved with increasing test generation time on 150 contracts

Generation Time	BF	GF	AF	GA
15 secs	45.1%	47.5%	52.4%	35.5%
30 secs	52.3%	55.8%	58.8%	47.9%
45 secs	57.7%	59.4%	65.1%	56.5%

contract with three dependent functions – `assign`, `approve`, and `send`. The `send` function sends money to the user address that is approved by the `approve` function which in turn checks if the address was allocated by the `assign` function. Thus for a test input to cover the send money transaction, the input will also need to have passed the condition checks in the `assign` and `approve` functions. Such a constraint dependent on other functions is not considered by any of the testing techniques.

3. Unused Functions in Inherited Contracts - Solidity language supports inheritance using the keyword `is`. Inherited contracts are found in both Random-C and Vulnerable-C datasets. We find not all inherited functions get called in the child contract. These uncalled functions contribute to the reduced coverage observed.

4. Modifier usage in the function - According to Solidity 0.6.2 documentation [31], "Modifiers can be used to change the behaviour of functions in a declarative way." Listing 6.3 shows a code snippet for a modifier function from the Bitconnect contract in the Random-C dataset that restricts calling of this function to the creator. None of the input generation techniques consider conditions imposed by function modifiers which in turn affects code coverage.

```
1 modifier onlyCreator() {
2     require(msg.sender == creator);
3     _;
4   }
```

Listing 6.3. onlyCreator modifier from the Bitconnect smart contract in Random-C dataset

Branch Coverage. Figures 6.3(d) and 6.4(d) show the histogram frequencies of branch coverage achieved by all four techniques over the Random-C and Vulnerable-C datasets, respectively. Median coverage is shown as a vertical dashed line. Median branch coverage for contracts in the Random-C dataset are 45% for AF, 43% for GF, 37% for BF and 35% for GA. Median values over the Vulnerable-C dataset are 49% for AF, 47% for GF, 46% for BF and 44% for GA. Reasons for uncovered branches observed with all the technqiues was discussed earlier in the context of uncovered code, namely, restricted

Table 6.2. P-values using One way Anova Tukey's HSD for pairwise comparison of coverage achieved for Random-C dataset

Comparison	Opcode	Call	Event	Branch
GA vs BF	**0.041**	**0.028**	**0.021**	**0.048**
GA vs GF	**0.023**	**0.011**	**0.008**	**0.031**
GA vs AF	**0.016**	**0.001**	**0.004**	**0.007**
GF vs BF	0.554	0.511	0.178	0.478
GF vs AF	0.057	0.079	0.417	0.067
BF vs AF	0.050	0.050	0.051	**0.044**

Table 6.3. P-values using One way Anova Tukey's HSD for pairwise comparison of coverage achieved for Vulnerable-C dataset.

Comparison	Opcode	Call	Event	Branch
GA vs BF	**0.049**	**0.001**	**0.065**	**0.049**
GA vs GF	**0.035**	**0.001**	0.073	**0.049**
GA vs AF	**0.008**	**0.001**	**0.014**	**0.011**
GF vs BF	0.719	0.758	0.571	0.763
GF vs AF	0.075	0.121	0.056	0.091
BF vs AF	0.050	0.205	0.061	0.099

input generation time, function dependencies and modifier usage. One-way Anova and Tukey's HSD test revealed significant difference between branch coverage achieved by fuzzing techniques (BF, GF and AF) versus the GA technique. A significant difference was also observed for BF versus AF over the contracts in the Random-C dataset, not the Vulnerable-C dataset. This is because nested conditions that BF struggles to cover occurs more frequently in the Random-C (29%) dataset than in Vulnerable-C (13%).

Opcode Coverage. Figure 6.3(a) and Figure 6.4(a) shows histogram frequencies of opcode coverage achieved by the four techniques over the Random-C and Vulnerable-C datasets, respectively. For the Random-C dataset, medians observed were 52% for AF, 49% for GF, 45% for BF and 36% for GA. Median coverage values over the Vulnerable-C dataset were – 60% for AF, 55% for GF, 53% for BF and 50% for GA.

Tables 6.2 and 6.3 show there is a significant difference between the fuzzing techniques versus the GA technique in opcode coverage achieved. On the other hand, pairwise comparison of AF, GF, BF revealed no sginificant difference.

It is worth noting for GF that several constraints provided to the Z3 solver for

uncovered branches provide no improvement in opcode coverage or remain unsolved. The first scenario of no improvement occurred with constraints for uncovered false branches of conditions with no code within, as seen in the following example.

```
function c(uint _0xbtcAmount) {
    some preparation...
    if (_0xbtcAmount != 0) {
        do something...
    }
}
```

Listing 6.4. btcToTokens function from the Bitconnect smart contract

Else statements are often omitted to limit gas usage in contracts. Many contracts with if-conditions resemble the example in the above listing, with no error-handling or code in the false branch. As a result, covering these additional branches do not result in additional opcodes being covered. The second scenario of constraints remaining unsolved occurred when conditions included Solidity specific features like modifiers, block number, blockhash, address data type and its member functions `send` and `transfer`, etc. Listing 6.5 shows an example function with an if condition that uses block related dependencies and address data type that Z3 fails to handle.

```
function clearApproval(address _owner, uint256 _tokenId)
    onlyOwner {
    if (ownerOf(_tokenId) == _owner && tokenApprovals[_tokenId] !=
        address(0) && tokenApprovals[_tokenId].fromBlock >
       _blockNumber ) {
      tokenApprovals[_tokenId] = address(0);
    }
}
```

Listing 6.5. clearApproval function from a smart contract in Random-C dataset

Z3 only solves 185 constraints from 54 smart contracts in our dataset. For these 54 smart contracts, opcode coverage increased from an average of 38.1% without using the solver to 54.3% with the solver.

Call Coverage. Figures 6.3(b) and 6.4(b) shows the call coverage frequency achieved by the four input generation techniques over the Random-C and Vulnerable-C datasets, respectively. Median call coverage values over the Random-C dataset are – 71% for AF, 68% for GF, 64% for BF and 54% for GA. Median call coverage values over contracts in the Vulnerable-C dataset are 78% for AF, 75% for GF, 75% for BF and 62% for GA.

We find fuzzing techniques achieve significantly better call coverage over contracts in both datasets than the GA technique. No significant difference was noted between pairs of fuzzing techniques.

Event Coverage. Figures 6.3(c) and 6.4(c) shows event coverage achieved by the testing techniques over the Random-C and Vulnerable-C datasets. Random-C median values were 67% for AF, 64% for GF, 59% for BF and 52% for GA. Vulnerable-C median event coverage was 73% for AF, 70% for GF, 70% for BF and 68% for GA.

As with other coverage criteria, difference between event coverage achieved by fuzzing techniques and GA was statistically significant. As seen in Figures 6.3 and 6.4, median values for event and call coverage are higher than those observed for opcode and branch coverage with all four techniques. This is because many of the event and call operations in a contract are not embedded within a conditional statement (around 55% for Random-C contracts) allowing them to be easily reached.

Summary. Over both datasets and for all four coverage metrics, AF achieves the best median performance among the four input generation techniques. Coverage achieved by all four techniques over contracts in both datasets is not that high, with medians ranging from 31% to 78% owing to small input generation time of 15 seconds per contract and not considering function dependencies and modifiers. Performance of GF could be improved by using an SMT solver that handles Solidity specific constructs.

Median values for coverage achieved over the Vulnerable-C dataset is slightly better than the Random-C dataset since the contracts are smaller on average (101 LOC in Vulnerable-C versus 389 LOC for Random-C) and contain fewer branches on average per contract (10 branches per contract for Vulnerable-C versus 22 for Random-C).

6.4.2 Q2. Fault Finding Comparison

We assess fault finding achieved by the four testing techniques using (1) Artificial seeded vulnerabilties in the Random-C dataset, and (2) Known vulnerabilities in the Vulnerable-C dataset.

6.4.2.1 Artificial Faults.

Figure 6.5 shows average mutation score for each technique grouped by mutation (or vulnerability) type. We seeded a single fault for each of the 10 different mutation types for each contract resulting in 16650 mutated contracts in our dataset. Across all mutated contracts, we find AF performs best at fault finding with an average mutation score

Table 6.4. P-values using One way Anova Tukey's HSD for pairwise comparison of mutation score.

Pairwise Comp.	Arithmetic	Div by Zero	Logical	Overflow	Relational	Rep. call	Timestamp	Tx. origin	Unchecked send	Underflow
GA vs BF	0.051	0.057	0.173	**0.005**	0.097	0.213	**0.004**	**0.017**	**0.044**	**0.039**
GA vs GF	**0.041**	**0.044**	**0.049**	**0.001**	0.074	0.179	**0.004**	**0.009**	**0.037**	**0.011**
GA vs AF	**0.037**	**0.041**	**0.036**	**0.001**	0.071	0.103	**0.001**	**0.001**	**0.033**	**0.002**
GF vs BF	0.144	0.471	0.365	0.515	0.736	0.716	0.469	0.148	0.214	**0.046**
GF vs AF	0.333	0.098	0.111	0.634	0.971	0.918	0.338	0.113	0.377	0.051
BF vs AF	0.071	0.115	0.332	0.071	0.741	0.883	0.377	0.092	0.125	**0.034**

of 72% versus 70% for GF (p-value = 0.39) and 67% for BF (p-value = 0.28) with no significant difference between them confirmed with one-way Anova followed by Tukey HSD test. However, AF was significantly better than GA with 56% average mutation score (p-value = 0.03). We also compared the average mutation score achieved by the four techniques for each of the different mutation types. P-values with one way Anova and Tukey HSD test is provided in Table 6.4 with emphasis on values showing significant difference (at 5% significant difference).

Overall, we find fuzzing techniques outperform the GA technique across all mutation types. As seen in Section 6.4.1, fuzzing techniques (AF, GF, BF) achieve better coverage and therefore execute more statements and operations than the GA technique. This in turn helps fuzzing techniques exercise and reveal more mutations than GA. For relational and repetitive function call mutation types, AF, BF, GF and GA have comparable perfomance with no significant difference, as none of them specifically target these constructs in the contract.

Finally, we observe 20–38% of the mutants remain alive even with the most effective testing technique, AF, in our experiment. This is because the generated inputs do not cover approximately 40% of the code and, as a result, fail to reveal seeded faults in these uncovered code regions. Reasons for regions of code being uncovered with the testing techniques was discussed in Section 6.4.1.

6.4.2.2 Real Faults.

Across all 90 vulnerable contracts in the Vulnerable-C dataset, we find AF performs best at vulnerability detection with an average score of 83% versus 78% for GF, 74% for BF and 56% for GA. One-way Anova test shows a significant difference between performance of AF and GA. Figure 6.6 shows average vulnerability detection score for each technique grouped by vulnerability type. AF has highest fault finding score for all vulnerability types, similar to our observation over artificial faults. All the fuzzing techniques achieve 100% detection score for the `timestamp` vulnerability. Presence of

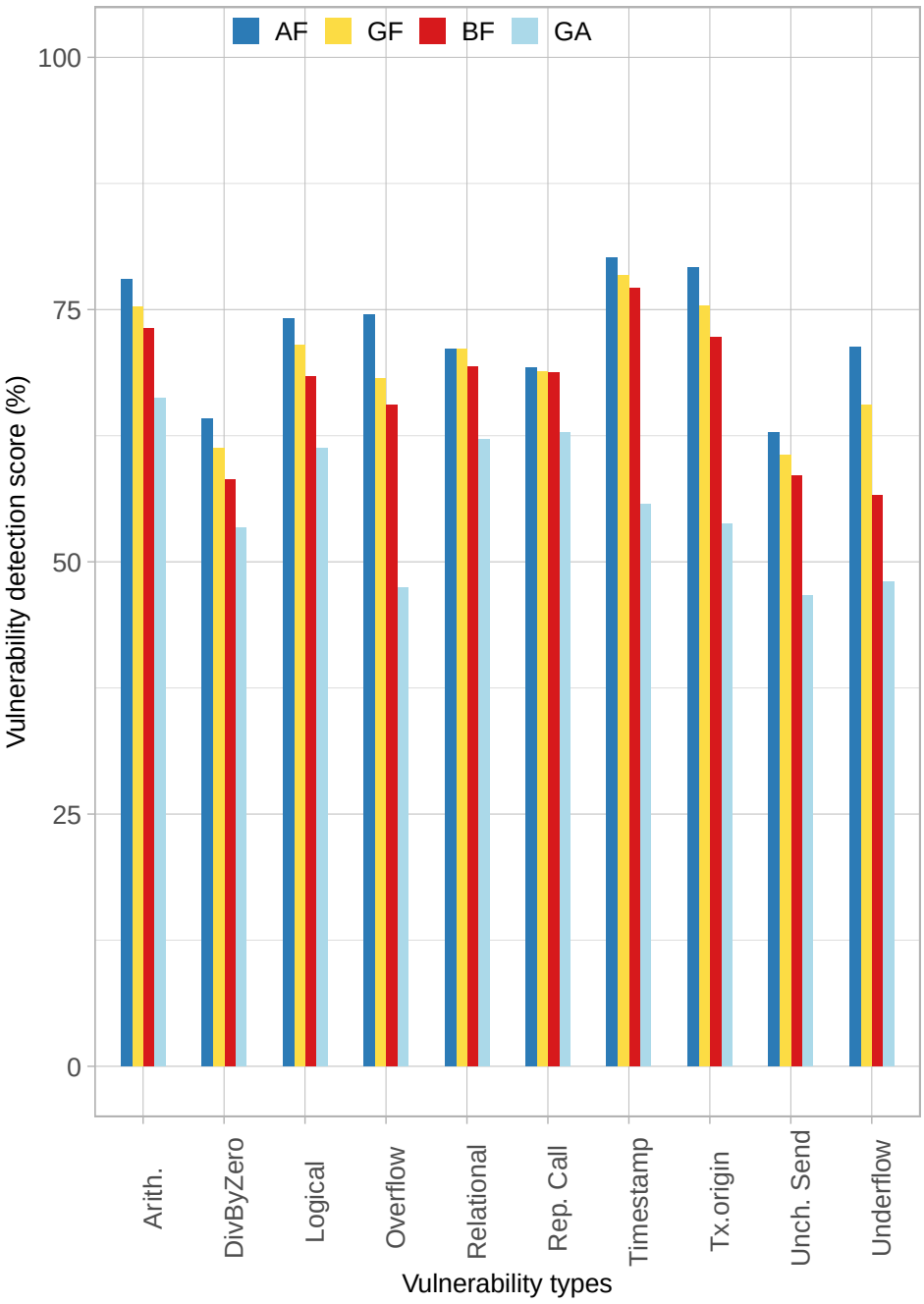

Figure 6.5. Fault finding score using 16650 artifical faults seeded in the Random-C dataset, grouped by mutation type

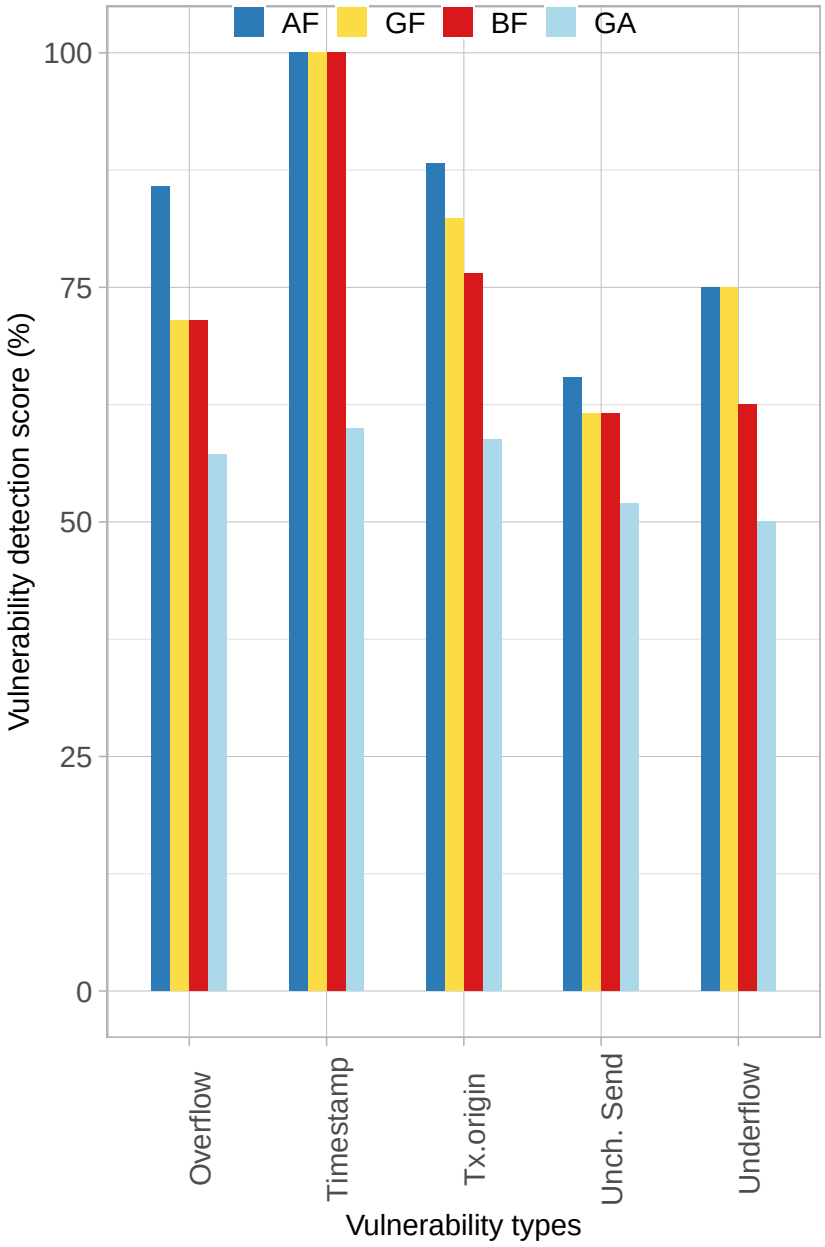

Figure 6.6. Fault finding score with 90 known vulnerabilities in the Vulnerable-C dataset, grouped by vulnerability type

this vulnerability in mostly small contracts (37 LOC on average) and lack of conditional statements around many of them made it easy to detect this vulnerability type.

We find all four techniques were unable to detect vulnerabilities in 17 out of the 90 contracts. This may be attributed to the following reasons, (1) Uncovered code - vulnerabilities in uncovered code remain undetected. As seen earlier, uncovered code is a pervasive issue across our contracts. (2) Branch conditions in contracts that involve Solidity specific variable types such as deployed address, balance, remain uncovered by test inputs from all four techniques. Listing 6.6 shows an example of an unchecked send vulnerability in the uncovered true branch of the if statement with Solidity specific features – block number and contract balance.

```
1    function finalize(uint _value) {
2    ...
3    if(this.balance> _value && allowed[from][msg.sender]>=_value
         && _value > 0 && Approval[from]._fromBlock > _blockNumber)
4        owner.send(_value);
5        // Unchecked send vulnerability
6    ...
7    }
```

Listing 6.6. finalize function from the MigrationAgent smart contract

Summary. We find fault finding scores for all four techniques is higher for real over artificial vulnerabilties. This is because all four testing techniques achieve better code coverage over contracts in the Vulnerable-C dataset compared to Random-C dataset.

6.4.3 Q3. Input set size and execution time of techniques.

For the Random-C and Vulnerable-C datasets, we present average size of the input sets generated by the four testing techniques and their execution times in Table 6.5. BF, as one might expect, generates the largest input set in the given time of 15 seconds - an average of 973 test inputs for Random-C contracts and 1321 test inputs for Vulnerable-C contracts. GF generates half as many test inputs as BF. This is because GF runs random input generation for half the generation time and the remaining time is used by the Z3 solver to generate inputs for uncovered branch conditions. The Z3 solver does not generate many test inputs as it was only able to solve constraints for 54 contracts. AF generates a much smaller input set, 47 test inputs and 27 test inputs on average, for contracts in the Random-C and Vulnerable-C datasets, respectively. This is because the AF algorithm reduces the input set size with respect to branch coverage after every

evolution step. Input set size for GA remains constant at 100 as that is the population size provided. Trends in execution time follow input set size with BF taking up most time and AF with least time. Overall AF generates the smallest input set on average with least overhead in execution.

Table 6.5. Average input set size and execution time per contract

Tech.	Average input set size		Average Execution Time (secs)	
	Random-C	Vulnerable-C	Random-C	Vulnerable-C
BF	973	1321	7.93 secs	9.02 secs
GF	436	587	4.31 secs	4.78 secs
AF	47	24	0.8 secs	0.31 secs
GA	100	100	1.78 secs	1.65 secs

6.5 Threats to Validity

A potential threat to the internal validity is bugs in implementation of the testing techniques or coverage measurement. We extensively tested our implementations and manually inspected them to mitigate this risk. Furthermore, the implementations for the testing techniques and coverage measurement along with the raw data are publicly available for other researchers and potential users to check the validity of our results.

A potential threat to the external validity is related to the fact that the set of smart contracts we have considered in this study may not be an accurate representation of a contract under test. We attempt to reduce the selection bias by leveraging a large collection of 1665 real, reproducible smart contracts. Additional threat to validity is caused by using artificially seeded faults to assess fault finding. To mitigate this threat, we also used a dataset of 90 vulnerable contracts, collected by Durieux et al. [32], deployed in the Ethereum network to assess fault finding. We also aim to reduce threats to external validity and ensure the reproducibility of our evaluation by providing the the scripts used to run the evaluation, and all data gathered.

6.6 Summary

In this chapter, we evaluated four smart contract input generation techniques - BF, GF, AF and GA by measuring,(1) Different types of code coverage - branch, event, call and opcode coverage, and (2) Fault finding - using real and artificially seeded vulnerabilities.

We used 2 datasets – Random-C, with 1665 contracts from Etherscan, and Vulnerable-C comprising 90 contracts with known vulnerabilities. The fuzzing techniques did significantly better than GA at achieving code coverage and fault finding for a fixed input generation time of 15 seconds per contract. AF did slightly better than BF and GF in terms of median coverage and fault finding values but the difference was not statistically significant. AF had the least overhead among the four techniques, generating the smallest input sets while being effective at covering the code and detecting vulnerabilities.

There is room for improvement for all four techniques in terms of code coverage and fault finding. We believe considering dependencies between functions and handling Solidity specific features will help improve the performance of these techniques considerably.

Chapter 7

Conclusion

This thesis presents novel techniques for automated test input generation and test effectiveness measurement for Solidity smart contracts. Our proposed techniques address challenges related to smart contract vulnerabilities, syntax and semantics, namely Solidity language-specific data and features, and the extent of vulnerability detection.

7.1 Summary of Contributions

This thesis makes four main contributions:

Instrumentation and Analysis Framework for Solidity Smart contracts

The thesis proposed SIF, a unified framework for Solidity contract analysis, query, instrumentation, and code generation to support Solidity smart contract developers and testers. SIF framework is capable of providing an interface to query the Abstract Syntax Tree (AST) of Solidity code. Also, it supports code instrumentation or transformation of the AST of Solidity contracts and generates Solidity code back. We empirically evaluated the SIF framework under 1838 smart contracts. We found that SIF was able to run all contracts.

Property Assertion Framework for Solidity Smart contracts

The thesis proposed SolAnalyser, an automated framework for vulnerability detection over Solidity smart contracts. This framework inserts suitable assert statements as pre and post conditions where needed. After inserting property checks into Solidity code, it

generates a smart contract with property checks. Finally, it generates automated test inputs for generated Solidity code to detect the vulnerabilities over the tested contract. To evaluate the SolAnalyser framework, we used 1838 verified smart contracts and 12866 mutated contracts which are generated by artificially seeding vulnerability. Our evaluation showed that the average precision and recall rate of our framework are 72% and 100%, respectively.

Novel Test Input Generation Techniques

In this thesis, we implemented different test input generation techniques. These are namely Blackbox fuzzing (BF), Greybox fuzzing with an SMT solver (GF), and Genetic Algorithm with a single objective (GA). The BF technique was implemented as a part of the SolAnalyser framework. GF and GA techniques were implemented in Java language. The evaluation of these techniques was done under two different datasets, contain 1755 smart contracts and a wide variety of the Solidity constructs, with respect to code coverage achieved and fault finding ability. Our experiments showed that Fuzzing based techniques were able to achieve better code coverage and fault-finding under our experiment setup.

Test Effectiveness Measurement

The thesis proposed a Coverage Environment (CovCal) and an Execution Environment (ExEn) to evaluate the effectiveness of the generated test inputs for Solidity smart contracts. CovCal measures the effectiveness of the test input based on our coverage metrics, including opcode, branch, call and event coverage, specifically designed for Solidity smart contracts. ExEn analyses the execution traces and report triggered vulnerabilities during the execution of a smart contract and measures fault finding capabilities of the test inputs. We conducted an empirical evaluation using 1755 Solidity smart contracts and 16650 mutated contracts, with their associated test inputs. Our evaluation showed that CovCal and ExEn were able to measure code coverage and fault-finding score for contracts under test.

7.2 Future Work

This section describes potential directions of future work.

Support for Smart Contracts Written in Vyper, Serpent and Liquidity

As mentioned in Section 2.2, Solidity is the most popular language to implement smart contracts on the Ethereum network. Therefore, the techniques we proposed in this thesis focus on Solidity language. However, there exist different programming languages, including Vyper, Serpent, Liquidity, to implement smart contracts. Vyper, Serpent and Liquidity are also high-level programming languages. Their structure is similar to Solidity; however, they have different keywords and notations. Therefore, we believe that mitigating our frameworks and testing techniques to Vyper, Serpent and Liquidity should involve only engineering efforts.

Support for Modifiers and Data flow Dependency

As mentioned in Section 6.4.1, according to our empirical evaluation on test input generation techniques, the main reasons for uncovered code regions are lack of support for data flow dependency and modifiers. We believe that executing functions considering dependency on other functions and considering conditions imposed by function modifiers will significantly increase the code coverage in smart contracts. Therefore, as future work, we plan to add these features to our input generation techniques.

7.3 Concluding Remarks

Although many attacks happened in the past and resulted in huge financial losses, the popularity and usage area of smart contracts are increasing sharply. As a result of language-specific features, particular run-time environment, and specific vulnerability types, Smart contracts require special testing techniques. This thesis proposes a novel approach to generate automated test inputs, measure the effectiveness of test inputs and detect the security vulnerabilities in the smart contracts.

Our experiment results on real-world smart contracts are promising. However, as mentioned in Section 7.2, there are further work to be done. In addition, diverse usage areas, evolving structures and programming models can pose new challenges in the future.

Despite its challenges, testing smart contracts is a fast-evolving and exciting area. The author of this thesis hopes that the proposed research and approaches will motivate developers and testers, as well as serve as a basis for future study on smart contract testing.

Bibliography

[1] M. A. Ahmed and I. Hermadi. Ga-based multiple paths test data generator. *Computers & Operations Research*, 35(10):3107–3124, 2008.

[2] S. Akca, C. Peng, and A. Rajan. Testing smart contracts: Which technique performs best? In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2021.

[3] S. Akca, A. Rajan, and C. Peng. Solanalyser: A framework for analysing and testing smart contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 482–489. IEEE, 2019.

[4] A. Alkhalifah, A. Ng, M. J. M. Chowdhury, A. Kayes, and P. A. Watters. An empirical analysis of blockchain cybersecurity incidents. In *2019 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pages 1–8. IEEE, 2019.

[5] F. E. Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[6] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin. Deploying search based software engineering with sapienz at facebook. In *International Symposium on Search Based Software Engineering*, pages 3–45. Springer, 2018.

[7] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[8] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[9] P. Asghari, A. M. Rahmani, and H. H. S. Javadi. Internet of things applications: A systematic review. *Computer Networks*, 148:241–261, 2019.

[10] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust*, pages 164–186. Springer, 2017.

[11] M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International conference on financial cryptography and data security*, pages 494–509. Springer, 2017.

[12] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.

[13] C. A. Bliss, M. R. Frank, C. M. Danforth, and P. S. Dodds. An evolutionary algorithm approach to link prediction in dynamic social networks. *Journal of Computational Science*, 5(5):750–764, 2014.

[14] F. Bond. A mutation testing tool for Solidity contracts . `https://github.com/federicobond/eth-mutants`,Accessed on: April 19, 2020.

[15] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[16] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[17] Brownie. Brownie. `https://github.com/iamdefinitelyahuman/brownie-v2`, Accessed on: August 24, 2020.

[18] V. Buterin. Vyper documentation. *Vyper by Example*, page 13, 2018.

[19] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[20] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[21] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang. scompile: Critical path identification and analysis for smart contracts. In *International Conference on Formal Engineering Methods*, pages 286–304. Springer, 2019.

[22] H. Chen, M. Pendleton, L. Njilla, and S. Xu. A survey on ethereum systems security: Vulnerabilities, attacks and defenses. *arXiv preprint arXiv:1908.04507*, 2019.

[23] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering*, 2021.

[24] M.-H. Chen, M. R. Lyu, and W. E. Wong. Effect of code coverage on software reliability measurement. *IEEE Transactions on reliability*, 50(2):165–170, 2001.

[25] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[26] L. W. Cong and Z. He. Blockchain disruption and smart contracts. *The Review of Financial Studies*, 32(5):1754–1797, 2019.

[27] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[28] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[29] C. Dannen. *Introducing Ethereum and solidity*, volume 318. Springer, 2017.

[30] G. A. Di Lucca and A. R. Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172–1186, 2006.

[31] S. Documentation. Solidity in Depth — Solidity 0.6.2 documentation. `https://docs.soliditylang.org/en/v0.6.2/solidity-in-depth.html`,Accessed on: February 25, 2021.

[32] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 530–541, 2020.

[33] I. Edwards. Parity technologies proposes solution to frozen ethereum wallets, 2019.

[34] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[35] eth sri. Security Scanner for Ethereum Smart Contracts. Accessed on: Sep.30, 2019.

[36] Etherscan. Etherscan - The Ethereum Block Explorer. `https://etherscan.io/txs`,Accessed on: April 19, 2020.

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

[38] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[39] G. Fraser and A. Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 33–36, 2016.

[40] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[41] M. Fu, L. Wu, Z. Hong, F. Zhu, H. Sun, and W. Feng. A critical-path-coverage-based vulnerability detection method for smart contracts. *IEEE Access*, 7:147327–147344, 2019.

[42] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

[43] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. Heimdahl. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3):1–34, 2016.

[44] A. Ghaleb and K. Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–427, 2020.

[45] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82, 2014.

[46] GovernMental. Ethereum contract 0xf45717552f12ef7cb65e95476f217ea008167ae3, 2016.

[47] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):116, 2018.

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

[49] D. J. Hand. Principles of data mining. *Drug safety*, 30(7):621–622, 2007.

[50] D. Harz and W. Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. *arXiv preprint arXiv:1809.09805*, 2018.

[51] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani. Smart contract vulnerability analysis and security audit. *IEEE Network*, 34(5):276–282, 2020.

[52] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[53] S. T. Howell, M. Niessner, and D. Yermack. Initial coin offerings: Financing growth with cryptocurrency token sales. Technical report, National Bureau of Economic Research, 2018.

[54] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jfuzz: A concolic whitebox fuzzer for java. 2009.

[55] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[56] B. Jiang, Y. Liu, and W. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269. ACM, 2018.

[57] N. Juristo, A. M. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1):7–44, 2004.

[58] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.

[59] S. Kim, J. A. Clark, and J. A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proc. Net. ObjectDays*, pages 9–12. Net. Objects Erfurt, Germany, 2000.

[60] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. Cab-fuzz: Practical concolic testing techniques for {COTS} operating systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 689–701, 2017.

[61] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[62] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[63] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[64] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[65] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[66] J. Li, B. Zhao, and C. Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

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

[68] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen, H. Song, and K.-K. R. Choo. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[69] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 814–819. IEEE, 2018.

[70] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Oyente: An Analysis Tool for Smart Contracts. Accessed on: Sep.30, 2019.

[71] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[72] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609, 2014.

[73] MAIAN-tool. MAIAN: automatic tool for finding trace vulnerabilities in Ethereum smart contracts. Accessed on: Sep.30, 2019.

[74] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[75] A. Miller, Z. Cai, and S. Jha. Smart contracts and opportunities for formal methods. In *International Symposium on Leveraging Applications of Formal Methods*, pages 280–299. Springer, 2018.

[76] C. Miller, Z. N. Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 4, 2007.

[77] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[78] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[79] MythX. Mythril Classic: Security analysis tool for Ethereum smart contracts. `https://github.com/ConsenSys/mythril-classic`,Accessed on: April 19, 2020.

[80] D. Nemin, M. Bernhard, and L. Wanseob. Ethereum Smart Contract Best Practices . `https://consensys.github.io/smart-contract-best-practices/known_attacks/`,Accessed on: January 30, 2021.

[81] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.

[82] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663. ACM, 2018.

[83] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.

[84] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2289–2306, 2020.

[85] S. Palladino. The parity wallet hack explained. *OpenZeppelin blog, https://blog. openzeppelin. com/on-the-parity-wallet-multisig-hack-405a8c12e8f7*, 2017.

[86] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[87] C. Peng. An automated mutation testing tool for C . `https://github.com/chao-peng/mutec`,Accessed on: May 10, 2020.

[88] C. Peng, S. Akca, and A. Rajan. Sif: A framework for solidity contract instrumentation and analysis. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–473. IEEE, 2019.

[89] C. Peng and A. Rajan. Cltestcheck: Measuring test effectiveness for gpu kernels. In *International Conference on Fundamental Approaches to Software Engineering*, pages 315–331. Springer, 2019.

[90] C. Peng and A. Rajan. Automated test generation for opencl kernels using fuzzing and constraint solving. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 61–70, 2020.

[91] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677. IEEE, 2020.

[92] V.-T. Pham, M. Böhme, and A. Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 543–553, 2016.

[93] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[94] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

[95] M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarrajan. A survey on automatic test case generation. *Academic Open Internet Journal*, 15(6), 2005.

[96] S. Registry. Smart contract weakness classification (swc) registry, 2018.

[97] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai. Empirical evaluation of smart contract testing: What is the best choice? 2021.

[98] sc forks. sc-forks/solidity-coverage. `https://github.com/sc-forks/solidity-coverage`, Accessed on: August 24, 2020.

[99] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 297–298, 2009.

[100] D. Siegel. Understanding the DAO attack. `http://www.coindesk.com/understanding-dao-hack-journalists`,Accessed on: April 19, 2020.

[101] SmartBugs. SmartBugs: A Framework to Analyze Solidity Smart Contracts. `https://smartbugs.github.io/`,Accessed on: August 24, 2020.

[102] SmartDec. SmartCheck. Accessed on: Sep.30, 2019.

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

[104] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[105] J. A. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[106] M. Swan. *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[107] N. Szabo. The idea of smart contracts. *Nick Szabo's Papers and Concise Tutorials*, 6, 1997.

[108] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018.

[109] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 103–119. IEEE, 2021.

[110] C. F. Torres, J. Schütte, and R. State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, 2018.

[111] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[112] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang. An overview of smart contract: architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 108–113. IEEE, 2018.

[113] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 2020.

[114] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[115] M. Wohrer and U. Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8. IEEE, 2018.

[116] G. Wood. A secure decentralised generalised transaction ledger [j]. *Ethereum project yellow paper*, 151:1–32, 2014.

[117] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[118] V. Wüstholz and M. Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020.

[119] Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.

[120] J. Ye, M. Ma, Y. Lin, Y. Sui, and Y. Xue. Clairvoyance: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 274–275. IEEE, 2020.

[121] M. Zalewski. American fuzzy lop (afl). *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[122] P. Zhang, F. Xiao, and X. Luo. Soliditycheck: Quickly detecting smart contract problems through regular expressions. *arXiv preprint arXiv:1911.09425*, 2019.