SolAnalyser: A Framework for Analysing and Testing Smart Contracts

SolAnalyser: A Framework for Analysing and Testing Smart Contracts

Sefa Akca
University of Edinburgh
Edinburgh, United Kingdom
s.akca@sms.ed.ac.uk

Ajitha Rajan
University of Edinburgh
Edinburgh, United Kingdom
arajan@ed.ac.uk

Chao Peng
University of Edinburgh
Edinburgh, United Kingdom
chao.peng@ed.ac.uk

***Abstract*—**
Executing, verifying and enforcing credible transactions on permissionless blockchains is done using smart contracts. A key challenge with smart contracts is ensuring their correctness and security. To address this challenge, we present a fully automated technique, SolAnalyser, for vulnerability detection over Solidity smart contracts that uses both static and dynamic analysis. Analysis techniques in the literature rely on static analysis with a high rate of false positives or lack support for vulnerabilities like out of gas, unchecked send, timestamp dependency. Our tool, SolAnalyser, supports automated detection of 8 different vulnerability types that currently lack wide support in existing tools, and can easily be extended to support other types. We also implemented a fault seeding tool that injects different types of vulnerabilities in smart contracts. We use the mutated contracts for assessing the effectiveness of different analysis tools.

Our experiment uses 1838 real contracts from which we generate 12866 mutated contracts by artificially seeding 8 different vulnerability types. We evaluate the effectiveness of our technique in revealing the seeded vulnerabilities and compare against five existing popular analysis tools – Oyente, Securify, Maian, SmartCheck and Mythril. This is the first large scale evaluation of existing tools that compares their effectiveness by running them on a common set of contracts. We find that our technique outperforms all five existing tools in supporting detection of all 8 vulnerability types and in achieving higher precision and recall rate. SolAnalyser was also faster in analysing the different vulnerabilities than any of the existing tools in our experiment.

***Keywords*—blockchain, smart contract, testing, static analysis, assertions, fault seeding**

I. Introduction

A blockchain is a distributed ledger that stores a growing list of unmodifiable records called blocks that are linked to previous blocks. Executing, verifying and enforcing credible transactions on blockchains is done using smart contracts [1]. 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 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 is still in question. For instance, failure of the contract, DAO [2], due to unsafe design choices resulted in losses of, approximately, $50 million. Many other vulnerabilities in contracts have been reported recently [3], [4], like the Fomo3D attack in 2018 that led to a loss of $4 million. Contracts can handle large numbers of virtual coins, which provides enough financial incentive for attacks. Unlike traditional distributed application platforms, contract platforms such as Ethereum [5] 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. Ethereum and Bitcoin allow network participants to decide which transactions to accept, how to order transactions, set the block timestamp, among others. Contracts which depend on any of these sources need to be aware of the subtle semantics of the underlying platform and explicitly guard against manipulation. In contrast to classical distributed applications that can be patched when bugs are detected, contracts are irreversible and immutable. There is no way to patch a buggy contract, regardless of its worth, without reversing the blockchain. This makes it critical to reason about the correctness of contracts *before* deployment.

Over the last couple of years, several techniques have been proposed for analysis of vulnerabilities in smart contracts [6]–[10]. Current solutions all rely on static analysis and symbolic execution to check the safety and correctness of smart contracts. These are complex, require significant analysis time, have restricted applicability (do not handle constructs like loops), a high rate of false positives, do not scale easily to large contracts and do not handle all types of well known vulnerabilties [11]. The techniques we propose in this paper 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 paper 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 paper is organised as follows. We present background on smart contracts and well known vulnerabilities in Section II. Related work in smart contract analysis is discussed in Section III. Our approach for static and dynamic analysis, test generation and fault seeding is discussed in Section IV. Experiment setup and results of our empirical evaluation is discussed in Sections V and VI, respectively.

II. Background

Our techniques for smart contract analysis targets the Ethereum blockchain, an open-source platform supporting smart contracts. A smart contract holds some amount of virtual coins (Ether), has its own private storage, and is associated with a predefined executable code. A contract state consists of two main parts: a private storage and the amount of virtual coins it holds (balance). Contract code can manipulate variables like in traditional imperative programs. 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, the most popular being Solidity [12] which is a JavaScript-like language, that is then compiled into EVM code using the `solc` compiler. To invoke a contract at an address, users send a transaction to that address. A transaction typically includes: payment to the contract for the execution and input data for the invocation.

On the EVM, each transaction is charged a certain amount of gas based on the operations within it. Aim of the gas system is to eliminate unnecessary operations in transactions. The gas fee schedule for the different operation types is published in the Ethereum yellow paper [5]. If the amount of gas required for transactions exceeds the gas limit provided by the contract owner, an out of gas exception is triggered.

A. Common Vulnerabilities

In this Section, we discuss eight well-known vulnerabilities that have limited support with existing tools and that are reported frequently in the smart contract weakness classification (SWC) registry [13]. Many of the other common vulnerabilities (32 others) are well supported by existing tools and we do not see the value in repeating analysis for them with our framework. A description of these other vulnerabilities and tools supporting them can be found in our report available at https://github.com/sefaakca/FirstyearReport.

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. Computations that exceed the size of the integer type result in overflow/underflow that may cause deviation from desired behaviour, making it a security vulnerability [8]. Integer overflow was recently detected in a token smart contract (BECToken) that erroneously allowed a large amount of tokens to be sent to receiver addresses in its Batch Transfer function.

Division by zero: Division by zero errors is another common cause of undesired behaviour in arithmetic operations in smart contracts. The Solidity compiler, Solidity version 0.4.15 onwards, can detect division by zero errors only if it can statically determine the divisor to be 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.

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 devious purposes. This timestamp dependence vulnerability was exploited in the GovernMental Ponzi scheme [14]. 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.

Authorisation through tx.orgin: `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 [13].

Unchecked send: A smart contract is able to call another contract with specific function calls 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 will not stop. If the call fails accidentally or an attacker forces the call to fail, it may cause unexpected behavior in the subsequent program logic [13].

Repetitive call function: According to Ethereum yellow paper [5], the call operation is a gas costly operation. Using call operations in a loop may result im undesired behavior such as Unexpected Ether balance or DoS With Block Gas Limit.
Out of gas: Every transaction is associated with an upper bound on the amount of gas that can be spent, and therefore the amount of computation allowed. This is the gas limit, mentioned earlier. If the gas spent exceeds this limit, the transaction will fail resulting in an out of gas exception. This vulnerability may be used in a Denial of Service attack.

III. Related Work

In the last few years, several analysis tools for detecting vulnerabilities in smart contracts have been proposed. We analysed twenty different existing tools in literature, namely, ContractFuzzer, EasyFlow, Maian, Echidna, Sereum, ECFChecker, Securify, SmartCheck, Zeus, Vandal, MadMax, Mythril, Manticore, Gasper, Porosity, Solgraph, Remix, Dr Y's Ethereum Contract Analyzer, F*, and Oyente. Sixteen of them use static analysis and four of them use dynamic analysis – namely EasyFlow, Echidna, Sereum and ECFChecker. In general, we find static analysis techniques are prone to high false alarm rates and analysis times. Dynamic analysis, on the other hand, may miss vulnerabilities as it depends on the quality of inputs used in execution and analysis. We discuss each of the 20 existing tools, vulnerability types supported, and their limitations in the following paragraphs.

ContractFuzzer [15] is a static analysis tool. It generates inputs for a smart contract with respect to the application binary interface file using a fuzzing technique. It supports exception disorder, re-entrancy, timestamp dependency, dangerous delegate call and freezing ether vulnerabilities. It is prone to a high rate of false alarms (or false positives if positive label is used for absence of bugs) for certain types of vulnerabilities such as timestamp dependency (62% reported in their paper [15]).

EasyFlow [16] is a taint analysis based tracking technique to monitor transactions. The tool only focuses on detecting integer overflow vulnerabilities. Maian [17] is a static analysis tool. It uses symbolic execution and concrete validation techniques to detect unchecked send, freezing ether and unchecked selfdestruct vulnerabilities. It has limited support for different vulnerability types. In our experiment in Section V, we find Maian is effective at detecting unchecked send vulnerability compared to existing tools.

Echidna [18] 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. Writing the predicates and test functions requires significant expertise and is time consuming.

Sereum [19] and ECFChecker [20] are dynamic analysis tools that focus on only detecting re-entrancy vulnerabilities. Securify [9] uses abstract interpretation to check violation patterns or compliance properties on the semantics extracted from the smart contract. We found in our experiment that Securify has a significant time overhead in analysis and also reports many false alarms.

SmartCheck [10] is a web-based analysis tool that translates Solidity code into an XML-based intermediate representation for analysis. We use SmartCheck in our experiment comparing existing tools. The vulnerabilities supported and its effectiveness is discussed in detail in Section VI. Zeus [8] uses abstract interpretation and symbolic model checking to analyse Solidity contracts. Zeus does not support all constructs in Solidity (like Throw, Self-destruct, virtual functions, assembly code block). The tool also reports high false positive rates for integer overflow/underflow errors and cannot detect division by zero.

Vandal [21] is a security analysis framework for smart contracts. It relies on symbolic analysis techniques. It translates Solidity bytecode to logic relations. It then checks the presence of vulnerabilities in translated logic relations. MadMax [22], built on top of the Vandal and Gasper [23] are static analysis tools. They check gas related vulnerabilities such as gas costly loops. Mythril [24] and Manticore [25] are other examples of analysis tools that use symbolic execution. The main drawback with these two tools is time complexity. We evaluated both tools with a small and simple contract, SafeMath [26] with 65 LOC. Mythril took 15 minutes and Manticore around 12 minutes to analyse this simple contract. Additionally, detection of integer overflow/underflow is associated with a high rate of false positives [9]. Mythril is discussed in detail in Section VI. Porosity [27], Solgraph [28] and Dr Y's Ethereum Contract Analyzer [29] are other tools to detect vulnerabilities. Dr Y's Ethereum Contract Analyser reflects to contract behaviour and checks the restriction of the contracts. Porosity checks re-entrancy vulnerability. Solgraph checks presence of unchecked send vulnerability. F* [6] is an analysis tool that translates Solidity code to F* language. The tool does not support structures like loops or struct that are commonly used in smart contracts. Oyente [7] is a symbolic execution tool that operates at the EVM byte code level to detect well known security issues like integer overflow and underflow, timestamp dependency and re-entrancy. It, however, has been shown to have a high rate of false positives [8]–[10]. The reasons for high false positive rate for Oyente is explained in Section VI in detail. Remix [30] is an online Solidity compiler and static analyser that is widely used by developers. Remix supports well known vulnerabilities like re-entrancy, timestamp usage, tx.origin usage, and out of gas. However, it cannot detect integer overflow/underflow and division by zero errors.

It is worth noting that two recent surveys [31], [32] compare analysis tools with respect to their vulnerability detection capability and discuss techniques used. However, these surveys simply take results presented in the original paper that have each been run on different contracts. This makes it impossible to objectively compare the tools. For the first time, we have taken a common set of contracts, seeded vulnerabilities, and checked the effectiveness of 5 existing tools in revealing the vulnerabilities and compared it against our technique, presented in Section V. Unlike existing tools in the literature that perform static analysis, our technique for vulnerability detection relies on both static and dynamic analysis. We believe, dynamic analysis will help reduce the number of false positives, time taken for analysis, and be widely applicable supporting all Solidity language features, vulnerability types, and large contract sizes.

IV. Approach and Implementation

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

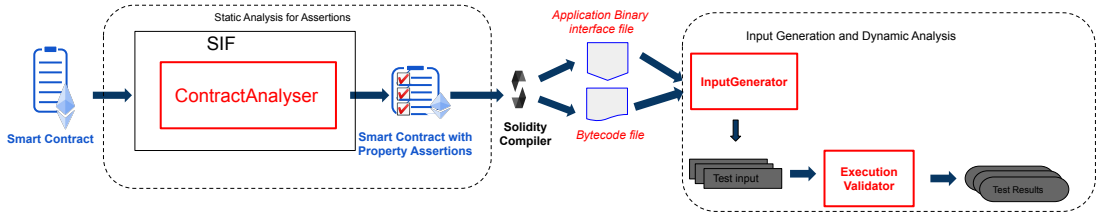

Fig. 1: SolAnalyser: Assertion injection, input generation and analysis

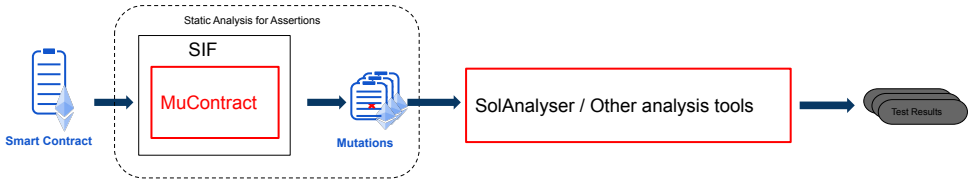

Fig. 2: MuContract: Artifically seed vulnerabilities in a Solidity Contract to produce Mutated/Buggy Contracts.

1) A vulnerability detection technique, `SolAnalyser`, shown in Fig. 1, that combines static analysis, implemented in `ContractAnalyser` using SIF [33] 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 VI.

We discuss each of these components in the following Sections.

A. 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. 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 [33]. 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 I 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/ContractAnalyser.

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/RandomInputGenerator.

Phase 3: Execution and analysis of instrumented contracts. The instrumented Solidity code with property assertions is executed in the Ethereum Virtual Machine (EVM) [34]. 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

TABLE I: 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

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. Cost of 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.

B. MuContract: Fault Seeding Tool

`MuContract` seen in Fig. 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 II. 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 II, 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/MuContract.

TABLE II: 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.

V. 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 II) 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 [15], 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`:

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 II. 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 [35] 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:

$$Precision = \frac{TP}{TP + FP}, \quad Recall = \frac{TP}{TP + FN} \qquad (1)$$

A. 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/Dataset. 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.

B. Oyente, Securify, Maian, SmartCheck and Mythril

We use the latest version of Oyente, Securify, Maian and Mythril provided in GitHub [24], [37]–[39] to analyse the 1838 contracts and 12866 mutated contracts. For analysis with SmartCheck, we load the smart contract into the web IDE [40] 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.

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

VI. Results and Discussion

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

A. 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 III. We find SolAnalyser 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.

B. 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. 3 and 4, respectively. Comparison with other tools shown in the figures is discussed in the next Section VI-C. 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.

C. 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

TABLE III: Extent of vulnerability support

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

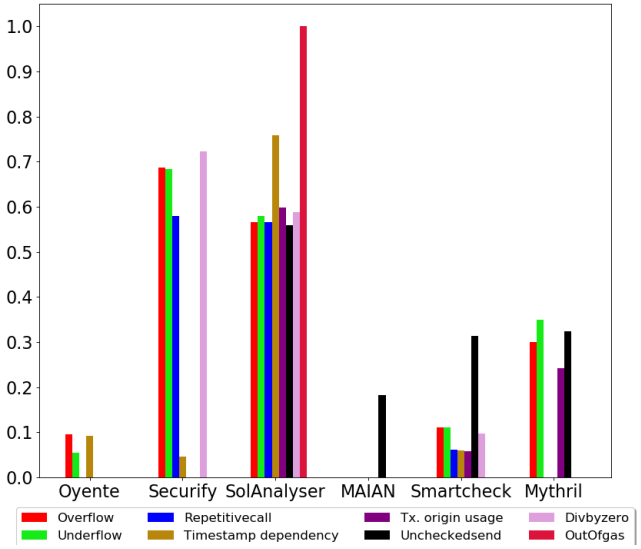

Fig. 3: Precision rate of Oyente, Securify, SolAnalyser, Maian, SmartCheck and Mythril for mutations

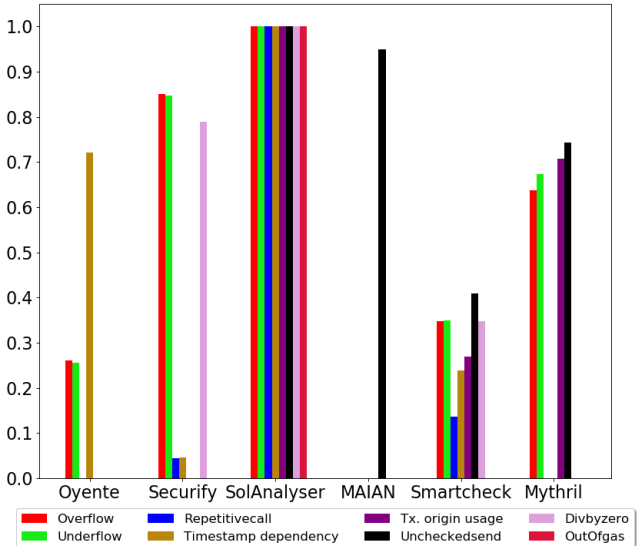

Fig. 4: Recall rate of Oyente, Securify, SolAnalyser, Maian, SmartCheck and Mythril for mutations

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 but has very poor precision (< 0.1) and recall (< 0.3). Oyente has a better Recall rate (0.72) in detecting timestamp dependency than integer overflow or underflow but misses timestamp dependences embedded in loops and multiple conditions.

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 access within `SolAnalyser`.

a) Summary.: Overall, 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.

b) 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.

VII. Conclusion and Future Work

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. In our future work, we plan to generate inputs that provide control flow coverage within functions and transactions in Solidity contracts. Finally, analysis time overhead is lowest for `SolAnalyser` (13.5 secs) compared to existing tools.

VIII. Acknowledgement

This work was supported in part by a grant from Huawei Innovation Research Program.

References

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

[2] D. Siegel, "Understanding the DAO attack," accessed on: Sep.30, 2019. [Online]. Available: http://www.coindesk.com/understanding-dao-hack-journalists

[3] 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.

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

[5] G. Wood, "A secure decentralised generalised transaction ledger [j]," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[6] 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*. ACM, 2016, pp. 91–96.

[7] 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*. ACM, 2016, pp. 254–269.

[8] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." NDSS, 2018.

[9] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," *arXiv preprint arXiv:1806.01143*, 2018.

[10] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," 2018.

[11] D. Cerezo Sánchez, "The valuation of secrecy and the privacy multiplier," 2018.

[12] Ethereum-Foundation, "The solidity contract-oriented programming language," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/ethereum/solidity

[13] SWC-Registery, "Smart Contract Weakness Classification and Test Cases," accessed on: Sep.30, 2019. [Online]. Available: https://smartcontractsecurity.github.io/SWC-registry/

[14] GovernMental, "Ethereum Contract," accessed on: Sep.30, 2019. [Online]. Available: https://www.etherchain.org/account/0xF45717552f12Ef7cb65e95476F217Ea008167Ae3\#code

[15] 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*. ACM, 2018, pp. 259–269.

[16] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "Easyflow: Keep ethereum away from overflow," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press, 2019, pp. 23–26.

[17] 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*. ACM, 2018, pp. 653–663.

[18] crytic, "Ethereum fuzz testing framework," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/crytic/echidna

[19] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[20] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 48, 2017.

[21] 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.

[22] 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*, vol. 2, no. OOPSLA, p. 116, 2018.

[23] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[24] MythX, "Mythril Classic: Security analysis tool for Ethereum smart contracts," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/ConsenSys/mythril-classic

[25] T. of Bits, "Manticore: Symbolic execution tool for analysis of smart contracts and binaries," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/trailofbits/manticore

[26] OpenZeppelin, "SafeMath from OpenZeppelin, a library for secure smart contract development," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol

[27] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF con*, vol. 25, p. 11, 2017.

[28] C. Raine Revere, Joaquim Pedro Antunes, "Visualize Solidity control flow for smart contract security analysis," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/raineorshine/solgraph

[29] H. Yoichi, "Dr. Y's Ethereum Contract Analyzer," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/pirapira/dry-analyzer

[30] Remix, "Remix documentation," accessed on: Sep.30, 2019. [Online]. Available: https://remix.readthedocs.io/en/latest/

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

[32] D. Perez and B. Livshits, "Smart contract vulnerabilities: Does anyone care?" *arXiv preprint arXiv:1902.06710*, 2019.

[33] C. Peng, S. Akca, and A. Rajan, "Sif: A framework for solidity code instrumentation and analysis," *arXiv preprint arXiv:1905.01659*, 2019.

[34] EthereumJS, "The Ethereum VM implemented in Javascript," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/ethereumjs/ethereumjs-vm

[35] D. J. Hand, "Principles of data mining," *Drug safety*, vol. 30, no. 7, pp. 621–622, 2007.

[36] Etherscan, "Etherscan - The Ethereum Block Explorer," accessed on: Sep.30, 2019. [Online]. Available: https://etherscan.io/txs

[37] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Oyente: An Analysis Tool for Smart Contracts," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/melonproject/oyente

[38] eth sri, "Security Scanner for Ethereum Smart Contracts," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/eth-sri/securify

[39] MAIAN-tool, "MAIAN: automatic tool for finding trace vulnerabilities in Ethereum smart contracts," accessed on: Sep.30, 2019. [Online]. Available: https://github.com/MAIAN-tool/MAIAN

[40] SmartDec, "SmartCheck," accessed on: Sep.30, 2019. [Online]. Available: https://tool.smartdec.net/