UNIVERSITY OF THESSALY

DEPARTMENT OF COMPUTER SCIENCE AND BIOMEDICAL INFORMATICS

ETHEREUM SMART CONTRACT TOOLS EVALUATION AND ANALYSIS

Computer Systems Security / Cybersecurity
Semester Project

Karantourou Paraskevi – Ioanna 01009
Douliaka Stefania 00974

INSTRUCTOR
SPATHOULAS GEORGIOS

CONTENTS

ABSTRACT

Blockchain is the most successful attempt in creating digital money- trust being the main factor of its success. Its ground-breaking technology can be simply described as a digital ledger of transactions that is duplicated and distributed across the entire network of computer systems. Each block in the chain contains a number of transactions, and every time a new transaction occurs on the blockchain, a record of that transaction is added to every participant's ledger. The decentralised database managed by multiple participants is known as Distributed Ledger Technology (DLT). One of the main elements of the blockchain is a smart contract. In the real world we use contracts, which can be defined as a written or spoken agreement, especially one concerning employment, sales, or tenancy. In blockchain we use smart (digital) contracts, which can be defined as self-executing elements with the terms of the agreement between buyer and seller being directly written into lines of code. Smart contracts are small programs written into blocks running on top of a blockchain that can receive and execute transactions autonomously without trusted third parties. Ethereum is the most popular framework for executing smart contracts. Of course, there can be no code without bugs and errors and bugs and errors existing in a piece of code concerning blockchain, when financial transactions are at stake, can be extremely dangerous. Multiple vulnerabilities are created and malicious assailants will seize every opportunity to take advantage of these flaws.

In the following pages we will present all the currently available tools, which are used to detect and eliminate said vulnerabilities Specifically, we are going to look into 86 security analysis tools developed especially for Ethereum blockchain, and emphasize in the forms of analysis of specific ones, all of them being publicly available under an open source license (we won't focus on publicly unavailable, neither as source nor as binary.)

1. INTRODUCTION

One can easily find more than enough articles referring to crypto attacks, where the outcome most of the time was loss of huge amounts of money. Transactions on Ethereum are immutable and cannot be reverted, so losses cannot be recovered. Bugs even occur in contracts by experienced programmers, which underlines the fact that smart contract programming is tricky. Therefore, numerous methods and tools have emerged to support the development of secure smart contacts and to aid the analysis of already deployed ones. Luckily, since 2016, developers are creating useful weapons to counterattack those threats. As mentioned above, these are called Ethereum Analysis Tools and there are 86 available, for now. Each tool works differently and applies on a different set of vulnerabilities.

2. BACKGROUND

2.1 VULNERABILITIES

Smart contracts are at the present time known-to-be-secure as well as known-to-be-vulnerable. Researchers have presented various types of these vulnerabilities, their number varying between 10 to 40. Some common types are:

Indirect execution of unknown code

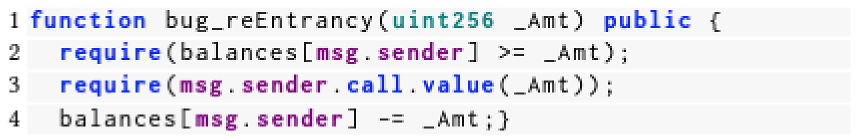
A possibility of indirect execution is there due to the presence of the fallback function feature in smart contracts. There are several reasons why this function can be called:

- Calling a function of another contract using ABI: if there is a typo in the signature string passed for encoding, or a function with such a signature does not exist, then the fallback function will be called;
- Deposit to another contract generates a call to its fallback function;
- Calling a function of another contract using the API: if the developer makes a mistake when declaring the interface of the called contract (for example, mixes up the type of a parameter), then the fallback function will be called.

Re-entrancy

In Ethereum, calls to functions of other smart contracts occur synchronously. That is, the calling code waits for the end of the execution of the external method before continuing its own work. This can cause the called contract to use the intermediate state of the calling contract.

This situation is not always obvious during development, if possible fraudulent actions on the part of the called contract are not taken into account.



```
function bug_reEntrancy(uint256 _Amt) public {
  require(balances[msg.sender] >= _Amt);
  require(msg.sender.call.value(_Amt));
  balances[msg.sender] -= _Amt;}
```

Figure 1: Example of Re-entrancy

Incorrect calculation of the output token amount

Most modern smart contracts deal with enormous amounts of money depicted in tokens or ETH value. Thus, a lot of operations in contract logic are connected with tokens' transfers to and from the contract. It creates a wide field for different mistakes connected to correct percentages, fees, and profits calculations. Top of such errors are:

- incorrect decimals handling;
- incorrect order of operation during fees calculations, which leads to the significant accuracy loss;
- Actually forgotten accuracy constant in the mat operations

All these errors lead to lost users' funds or even tokens locked forever.

Interface / naming issues

An example of this is the Rubixi smart contract. Rubixi was a Ponzi game, in which its owner could transfer the fees accumulated in the financial pyramid to himself. Usually, the owner is set in the constructor of the contract, which is called when it is created, the same logic was implemented in the Rubixi smart contract. It should be noted that in the Solidity versions prior to 0.4.22, constructors were defined as functions with the same name as the name of the contract. At some point, the contract was renamed from Dynamic Pyramid to

Rubixi, but the developer forgot to change the name of the constructor, so anyone who called the Dynamic Pyramid function could become the owner of a contract and could steal the accumulated funds.

Dependency on the order of execution

The state of a contract is determined by the values of its variables, which are changed by calling its functions. Calling a smart contract function is the same transaction as a transaction of ETH or ERC-20 token transfer. These transactions are finalized by the network only after the next block creation is complete. Thus, when a user sends a transaction to call a contract function, he cannot be sure that the transaction will be executed in the same state of the contract in which he was at the time of sending. This can happen because other transactions in the same block have changed the state of the contract. Moreover, miners have some freedom in ordering transactions when forming a block, as well as in choosing to include a particular transaction in a block. In some cases, the impossibility of determining the state of the contract, in which the transaction will be executed, can cause vulnerability of the contract itself. It also becomes especially dangerous to interact with contracts written in such a way that their behaviour can be changed over time.

Time component

Sometimes the logic of smart contracts can be time-dependent. The time for a contract is only available in the context of a transaction. The timestamp of a transaction, in turn, is equal to the label of the block in which it is included. Thus, consistency with the state of a smart contract is achieved. However, this also creates an opportunity for the miner to abuse his/her position due to some freedom in setting a timestamp for the block. So, the miner has some advantage over other parties to a contract that he/she could exploit to his/her own benefit.

Using the blockhash function

Using the blockhash function is similar to the timestamp dependency, it is not recommended to use it for important components for the same reason as with the timestamp dependency, because miners can manipulate these functions and change the withdrawal of funds in their own favour. This is especially noticeable when block hash is used as a source of randomness.

Incorrectly handled exceptions

There are many situations where an exception can be thrown in Solidity, but the way these exceptions are handled is not always the same. Exception handling is based on interactions between contracts. Thus, contracts are vulnerable to attack from malicious users, if these exceptions are not handled properly, transactions will be rolled back.

Incorrect work with ERC20 token

There is a well-known OpenZeppelin (for building secure smart contracts) implementation of the ERC-20 token (standard used for creating and issuing smart contracts), overused in modern protocols. In most cases, it is completely applicable, and its functionality is enough for correct financial operations. Though there is a place for custom implementations of a token standard. Thus, it creates a place for discrepancies between the newly created token and actual ERC20 standard – small inconsistencies like missing return value in transfer() function. Though, such a small change may lead to the non-functional method of the contract, since it will not be able to recognize the interface. It is a very tiny mistake, almost not noticeable during testing, but in production, it leads to stuck funds and blocked contracts.

Gas Related Issues

There can be several gas-related issues like sending a transaction with insufficient gas, useless code in the smart contract, or gas costly loops present in the contract. Gas is used as a transnational fee to execute instructions of the smart contract like each type of operation requires a different gas, which is charged in Ether (Wei-smallest unit of Ether).

Needless to say, this is only the tip of the iceberg regarding the vulnerabilities that can appear. Each analysis tool, can solve one or more of these problems and each one can work and be used in a different way. Due to their diversity, there are many ways to categorize them and perhaps the most common one is between static and dynamic analysis tool. Static code analysis is a process to debug the code by examining it without actually executing the code, whereas dynamic code analysis is the analysis of the code that is performed by executing it on a real or virtual processor.

2.2 ANALYSIS TOOLS

Following, 86 analysis tools are presented. We will emphasize on the analysis of four of them, the ones that appear to be the most common smart contract auditing tools (Securify, Mythril, Oyente, Slither), according to the users and researchers.

1. **ContractWard** (static) year: 2019	**Input:** Solidity
2. **Echidna** (static) 2020	**Input:** Solidity or Viper code
3. **Eth2Vec** (static) 2021	**Input:** Solidity
4. **Ethainter** (static) 2020	**Input:** Solidity
5. **EthVer** (static) 2020	**Input:** Solidity
6. **FEther** (static) 2019	**Input:** Solidity
7. **FSPVM** (static) 2020	**Input:** Solidity
8. **GasGauge** (static) 2021	**Input:** Solidity
9. **Gastap** (static) 2018	**Input:** EVM byte code or disassembled EVM byte code or solidity code.
10. **MuSc** (static) 2019	**Input:** Solidity
11. **NeuCheck** (static) 2019	**Input:** Solidity
12. **Pakala** (static) 2018	**Input:** Solidity
13. **Remix IDE** (static) 2016	**Input:** Solidity
14. **SASC** (static) 2018	**Input:** Solidity
15. **sCompile** (static) 2018	**Input:** Solidity
16. **SESCon** (static) 2021	**Input:** Solidity
17. **SIF** (static) 2019	**Input:** Solidity
18. **SmartAnvil** (static) 2018	**Input:** Solidity
19. **SMARTBUGS** (static) 2020	**Input:** Solidity

20. **SmartCheck** (static) 2017	**Input:** Solidity
21. **SmartEmbed** (static) 2019	**Input:** Solidity
22. **Smart-Graph** (static) 2021	**Input:** Solidity
23. **SmartInspect** (static) 2018	**Input:** Solidity
24. **SmartScan** (static and dynamic) 2021	**Input:** Solidity
25. **SolAnalyzer** (static) 2019	**Input:** Solidity
26. **Solc-Verify** (static) 2019	**Input:** Solidity
27. **Solgraph** (static) 2016	**Input:** Solidity
28. **SolGuard** (static) 2021	**Input:** Solidity
29. **Solhint** (static) 2017	**Input:** Solidity
30. **Solidifier** (static) 2020	**Input:** Solidity
31. **Soliditycheck** (static) 2019	**Input:** Solidity
32. **SolMet** (static) 2018	**Input:** Solidity
33. **VeriSmart** (static) 2020	**Input:** Solidity
34. **VeriSol** (static) 2019	**Input:** Solidity
35. **VeriSolid** (static) 2019	**Input:** Solidity
36. **Zeus** (static) 2018	**Input:** Solidity
37. **DEFECTCHECKER** (static) 2021	**Input:** EVM Byte Code (Ethereum Virtual Machine Byte Code)
38. **E-EVM** (static) 2018	**Input:** EVM Byte Code
39. **Erays** (static) 2018	**Input:** EVM Byte Code
40. **ESCORT** (static) 2021	**Input:** EVM Byte Code
41. **Ether (S-GRAM)** (static) 2018	**Input:** EVM Byte Code
42. **EtherTrust** (static) 2018	**Input:** EVM Byte Code
43. **EthIR** (static) 2018	**Input:** EVM Byte Code
44. **eThor** (static) 2020	**Input:** EVM Byte Code
45. **EthPloit** (static) 2020	**Input:** EVM Byte Code
46. **GasChecker** (static) 2020	**Input:** EVM Byte Code
47. **Gasper** (static) 2017	**Input:** EVM Byte Code
48. **HoneyBadger** (static) 2019	**Input:** EVM Byte Code
49. **KEVM** (static) 2016	**Input:** EVM Byte Code
50. **MadMax** (static) 2018	**Input:** EVM Byte Code
51. **Octopus** (static) 2017	**Input:** EVM Byte Code
52. **Osiris** (static) 2018	**Input:** EVM Byte Code
53. **Porosity** (static) 2017	**Input:** EVM Byte Code
54. **RA** (static) 2020	**Input:** EVM Byte Code
55. **Rattle** (static) 2018	**Input:** EVM Byte Code
56. **SmartSheild** (static) 2020	**Input:** EVM Byte Code
57. **TEEther** (static) 2018	**Input:** EVM Byte Code
58. **Vandal** (static) 2018	**Input:** EVM Byte Code
59. **VerX** (static) 2020	**Input:** EVM Byte Code
60. **Conkas** (static) 2019	**Input:** EVM Byte Code or Solidity
61. **GASOL** (static) 2020	**Input:** EVM Byte Code or Solidity
62. **SAFEVM** (static) 2019	**Input:** EVM Byte Code or Solidity
63. **FSolidM** (static) 2017	**Input:** Form. Spec

64. **ContractLarva** (dynamic) 2017	**Input:** Solidity
65. **Ethlint** (dynamic) 2016	**Input:** Solidity
66. **Harvey** (dynamic) 2020	**Input:** Solidity
67. **ModCon** (dynamic) 2020	**Input:** Solidity
68. **Solitor** (dynamic) 2018	**Input:** Solidity
69. **Vultron** (dynamic) 2019	**Input:** Solidity
70. **ContractFuzzer** (dynamic) 2018	**Input:** EVM Byte Code
71. **ContractGuard** (dynamic) 2019	**Input:** EVM Byte Code
72. **EthBMC** (dynamic) 2020	**Input:** EVM Byte Code
73. **Etherolic** (dynamic) 2020	**Input:** EVM Byte Code
74. **EVMFuzzer** (dynamic) 2019	**Input:** EVM Byte Code
75. **MAIAN** (dynamic) 2018	**Input:** EVM Byte Code
76. **Manticore** (dynamic) 2017	**Input:** EVM Byte Code
77. **Sereum** (dynamic) 2019	**Input:** EVM Byte Code
78. **sFuzz** (dynamic) 2020	**Input:** EVM Byte Code
79. **SODA** (dynamic) 2020	**Input:** EVM Byte Code
80. **EasyFlow** (dynamic) 2019	**Input:** EVM Byte Code or Solidity
81. **ReGuard** (dynamic) 2018	**Input:** EVM Byte Code or Solidity
82. **SoliAudit** (dynamic) 2019	**Input:** EVM Byte Code or Solidity
83. **Securify** (static) 2018	**Input:** EVM Byte Code
84. **Oyente** (static) 2016	**Input:** EVM Byte Code
85. **Mythril** (static) 2017	**Input:** EVM Byte Code
86. **Slither** (static) 2018	**Input:** Solidity

3. ANALYSIS

Characteristics to assess the overall performance of the tools:

- The number of false-positive vulnerabilities.
- The number of false-negative (missed) vulnerabilities.
- Audit runtime.
- Ability to find ways to optimise gas consumption.
- Additional functionality and digitalisation of obtained results.

The analysis proceeds in three main steps:

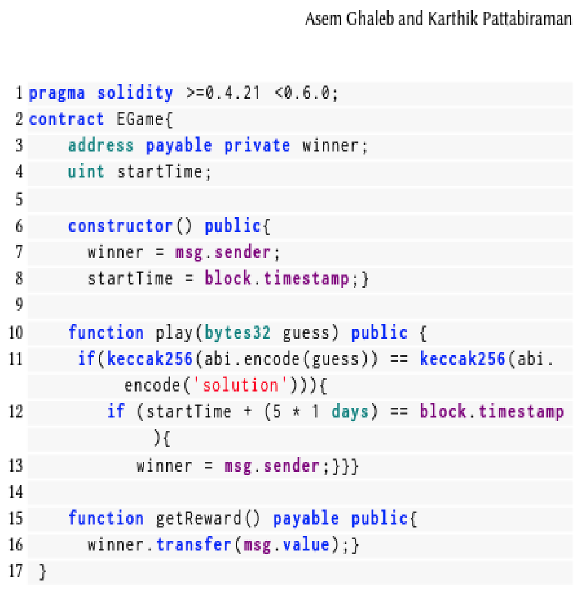
1. Parsing the EVM bytecode and decompiling it into a form suitable for analysis.
2. Identifying semantic facts about the smart contract.
3. Combining patterns and rules to identify vulnerabilities.

3.1 SMART CONTRACT AUDITING

Smart Contracts are written in a high-level language, such as Solidity. Solidity works on most platforms and is build for Ethereum Virtual Machine (EVM) Bytecode,, where the compile gets place to. EVM is deployed and stored in the blockchain accounts. One example of a methodology used to discover undetected security bugs is through bug injection, with systematic evaluation in order to find potential flaws. SolidiFI (Solidity Fault Injector) is an automated and systematic framework and is used to evaluate smart contract static analysis tools. The evaluation results show several cases of bugs that were not detected by the evaluated tools even though those undetected bugs are within the detection scope of tools. This method shows important flaws in the contracts, and provides a set of tests for the smart contract developers. Future work will consist of expanding SolidiFI to other smart contract languages than Solidity, and automating the bug definition processes for injecting new bug types, making the smart contracts even smarter.

<u>How it works</u>: SolidiFI injects bugs formulated as code snippets into all possible locations into a smart contract's source code written in Solidity. The code snippets are vulnerable to specific security vulnerabilities that can be exploited by an attacker. The resulting buggy smart contracts are then analyzed using the static analysis tools being evaluated, and the results are inspected for those injected bugs that are not detected by each tool. These are the false-negatives of the tool. Tools that work on the EVM bytecode will compile the buggy contracts to produce the EVM code for analysis. Then, the injected code is scanned using the static analysis tools. Finally, the results of each tool are checked, and false negatives and false positives are measured.

In the prior work of A. Ghaleb and K.Pattabiraman there is an illustration of a smart contract through a running example, adopted from a previous research [Atzeietal.2017].
This contract implements a public game that enables users to play a game and submit their guesses or solutions for the game along with some amount of money. The money will be transferred to the account of the last winner if the guess is wrong; otherwise the user will be set as the current winner and will receive the money from users who play later. The constructor() at line 6 runs only once when the contract is created, and it sets the initial winner to the owner of the contract defined by the user who submitted the create transaction of the contract (msg.sender). It also initializes the startTime variable to the current timestamp during the contract creation. The function play at line 10 is called by the user who wants to submit his/her guess, and it compares the received guess with the true guess value. If the comparison is successful, it sets the winner to the address of the user account who called this function, provided the guess was submitted within 5 days of creating the contract. Finally, the function getReward sends the amount of ether specified in the call to getReward (msg.value), to the last winner.

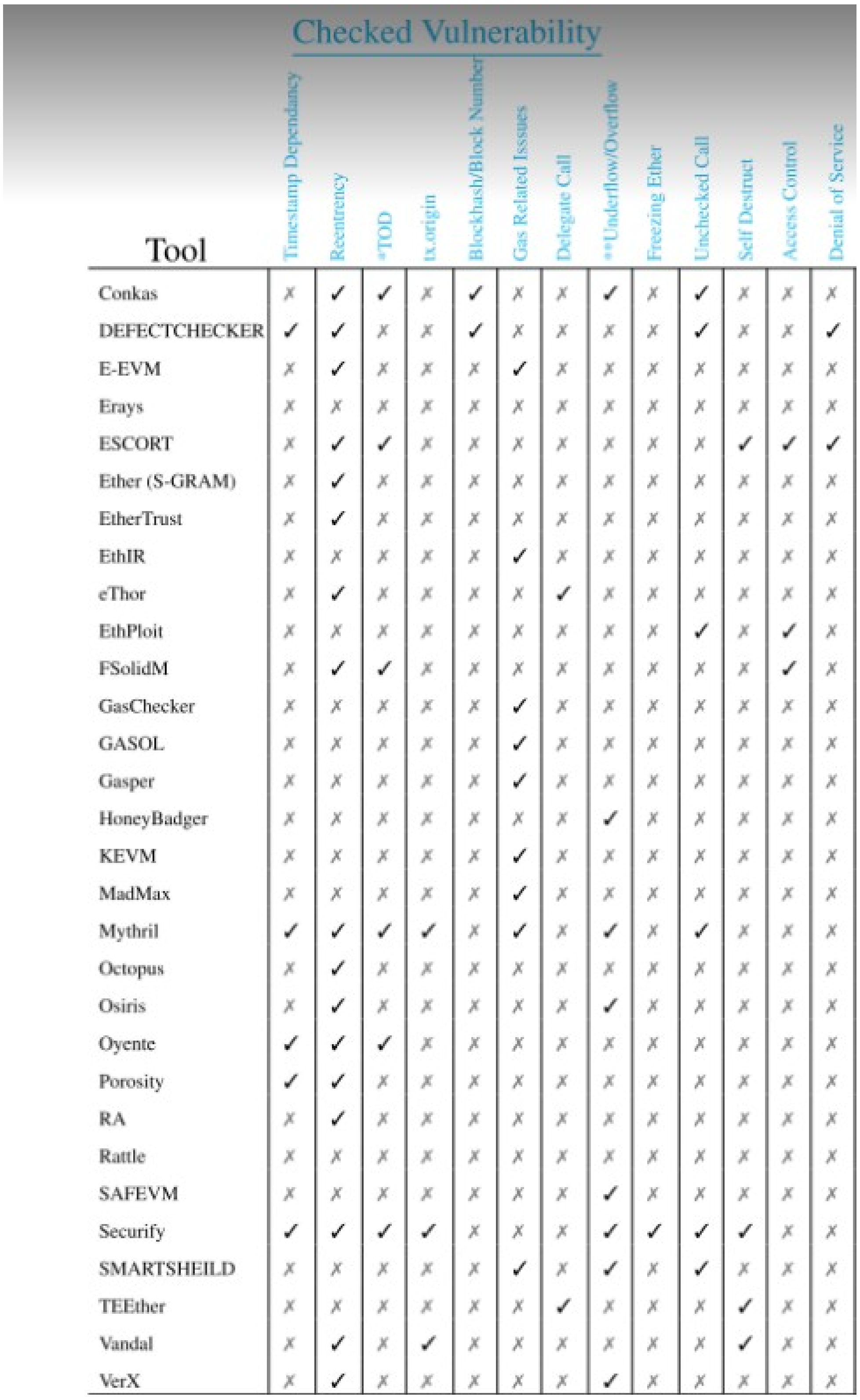

Asem Ghaleb and Karthik Pattabiraman

```
1 pragma solidity >=0.4.21 <0.6.0;
2 contract EGame{
3     address payable private winner;
4     uint startTime;
5
6     constructor() public{
7       winner = msg.sender;
8       startTime = block.timestamp;}
9
10    function play(bytes32 guess) public {
11     if(keccak256(abi.encode(guess)) == keccak256(abi.
          encode('solution'))){
12        if (startTime + (5 * 1 days) == block.timestamp
             ){
13           winner = msg.sender;}}}
14
15    function getReward() payable public{
16      winner.transfer(msg.value);}
17  }
```

Figure 2: Simple Contract written in Solidity

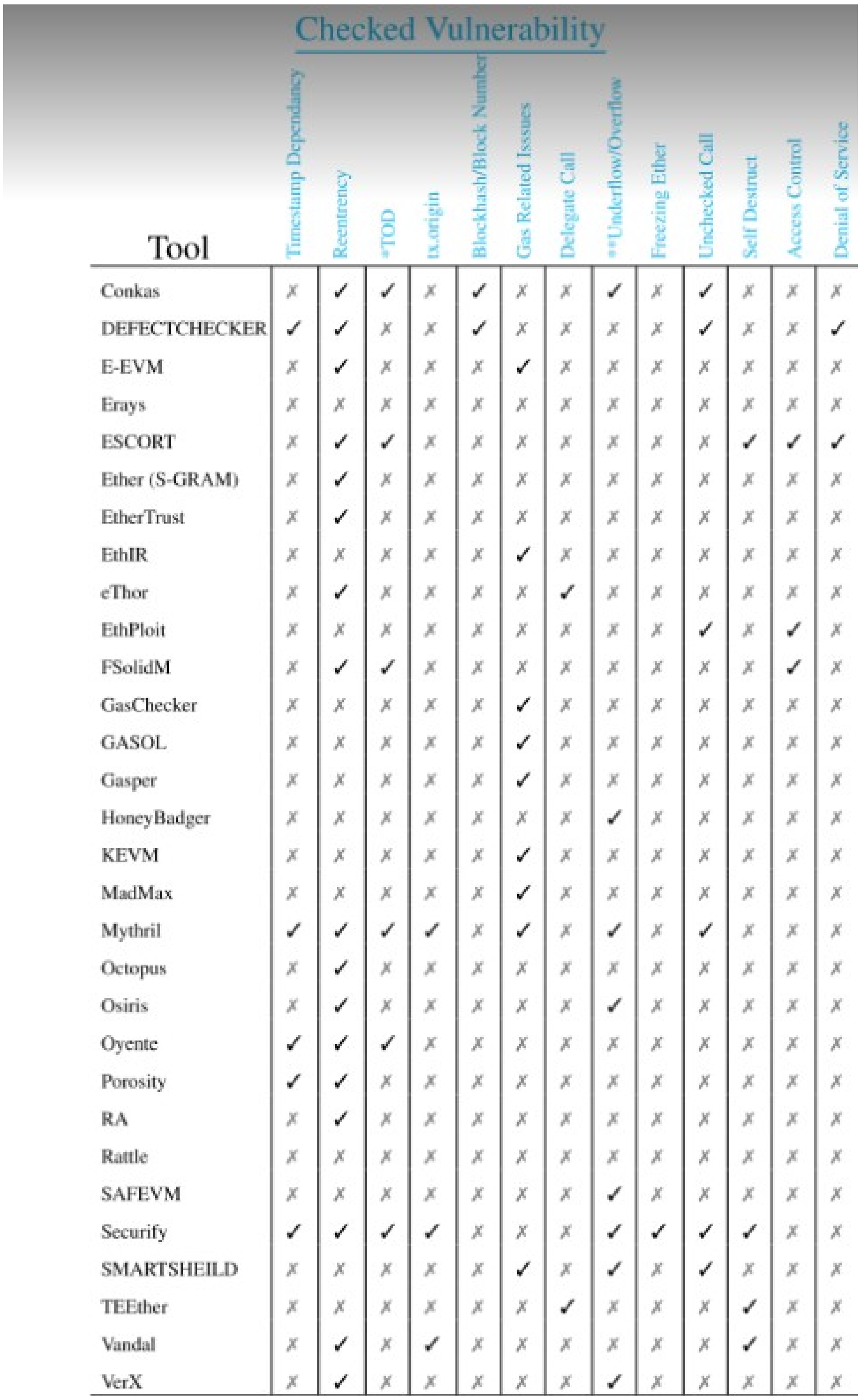
3.2 VULNERABILITY TOOLS AND AUDITING TEST

The following tables represent which vulnerabilities each tool can counterattack. Securify is the most advanced tool regarding formal guarantees. The Securify tool gives good results in a relatively short time. It has 37 templates for different types of vulnerabilities in its arsenal, which makes this tool stand out from its competitors. Securify can detect simple logic errors, but it cannot find ways to save and optimise gas costs.



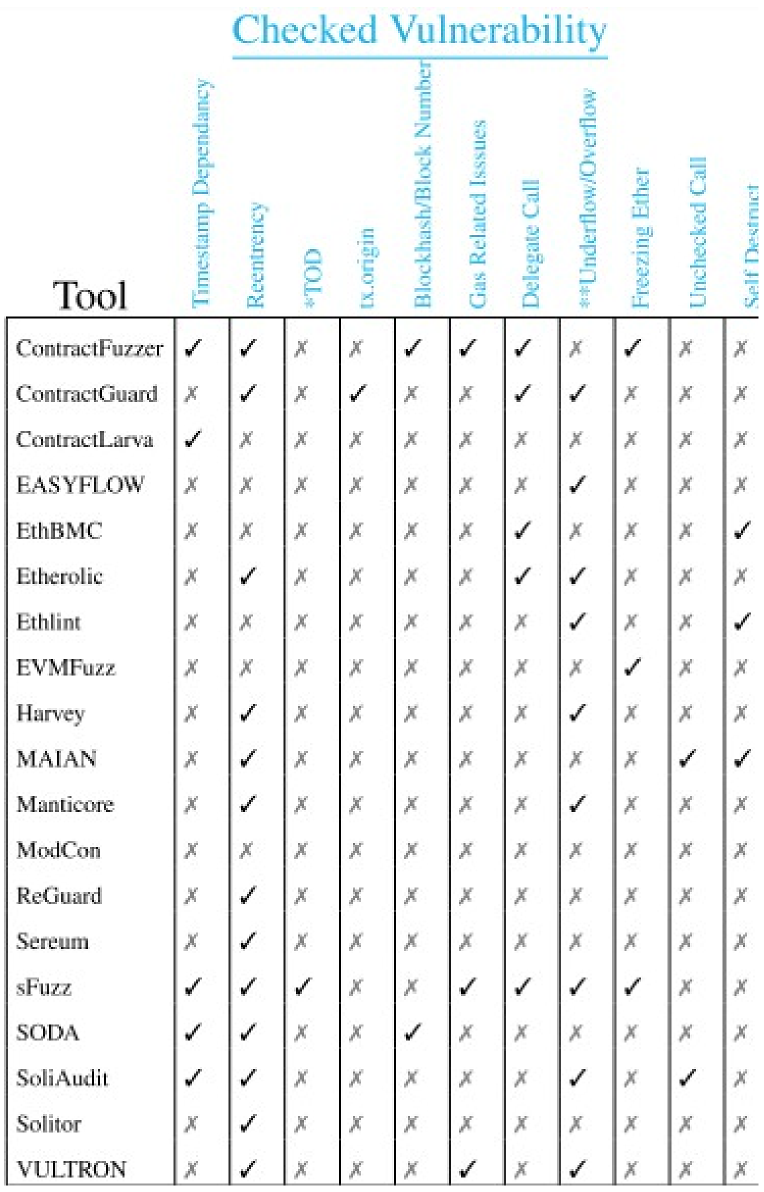
	Checked Vulnerability												
Tool	Timestamp Dependancy	Reentrency	*TOD	tx.origin	Blockhash/Block Number	Gas Related Isssues	Delegate Call	**Underflow/Overflow	Freezing Ether	Unchecked Call	Self Destruct	Access Control	Denial of Service
Conkas	✗	✓	✓	✗	✓	✗	✗	✓	✗	✓	✗	✗	✗
DEFECTCHECKER	✓	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓
E-EVM	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Erays	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ESCORT	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Ether (S-GRAM)	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
EtherTrust	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
EthIR	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
eThor	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
EthPloit	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗
FSolidM	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
GasChecker	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
GASOL	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Gasper	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
HoneyBadger	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
KEVM	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
MadMax	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Mythril	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗
Octopus	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Osiris	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Oyente	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Porosity	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
RA	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Rattle	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
SAFEVM	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Securify	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗
SMARTSHEILD	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗
TEEther	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗
Vandal	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
VerX	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗

Figure 3: Static Analysis



	Checked Vulnerability												
Tool	Timestamp Dependancy	Reentrency	*TOD	tx.origin	Blockhash/Block Number	Gas Related Isssues	Delegate Call	**Underflow/Overflow	Freezing Ether	Unchecked Call	Self Destruct	Access Control	Denial of Service
Conkas	✗	✓	✓	✗	✓	✗	✗	✓	✗	✓	✗	✗	✗
DEFECTCHECKER	✓	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✓
E-EVM	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Erays	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ESCORT	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Ether (S-GRAM)	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
EtherTrust	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
EthIR	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
eThor	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
EthPloit	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗
FSolidM	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
GasChecker	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
GASOL	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Gasper	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
HoneyBadger	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
KEVM	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
MadMax	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Mythril	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗
Octopus	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Osiris	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Oyente	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Porosity	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
RA	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Rattle	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
SAFEVM	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Securify	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗
SMARTSHEILD	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗
TEEther	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗
Vandal	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
VerX	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗

Figures 3 & 4: Static Analysis



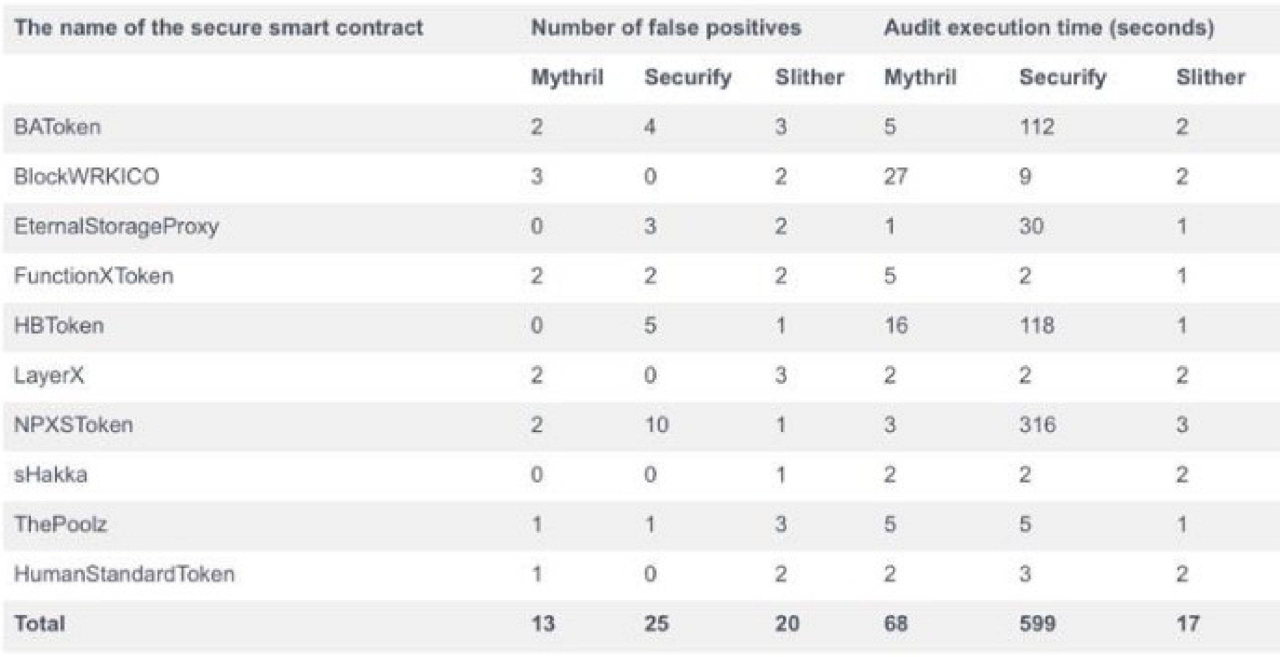
	Checked Vulnerability										
Tool	Timestamp Dependancy	Reentrency	*TOD	tx.origin	Blockhash/Block Number	Gas Related Isssues	Delegate Call	**Underflow/Overflow	Freezing Ether	Unchecked Call	Self Destruct
ContractFuzzer	✓	✓	✗	✗	✓	✓	✓	✗	✓	✗	✗
ContractGuard	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗
ContractLarva	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
EASYFLOW	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
EthBMC	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
Etherolic	✗	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Ethlint	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
EVMFuzz	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
Harvey	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
MAIAN	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
Manticore	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
ModCon	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ReGuard	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Sereum	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
sFuzz	✓	✓	✓	✗	✗	✓	✓	✓	✓	✗	✗
SODA	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
SoliAudit	✓	✓	✗	✗	✗	✗	✗	✓	✗	✓	✗
Solitor	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
VULTRON	✗	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗

Figure 5: Dynamic Analysis

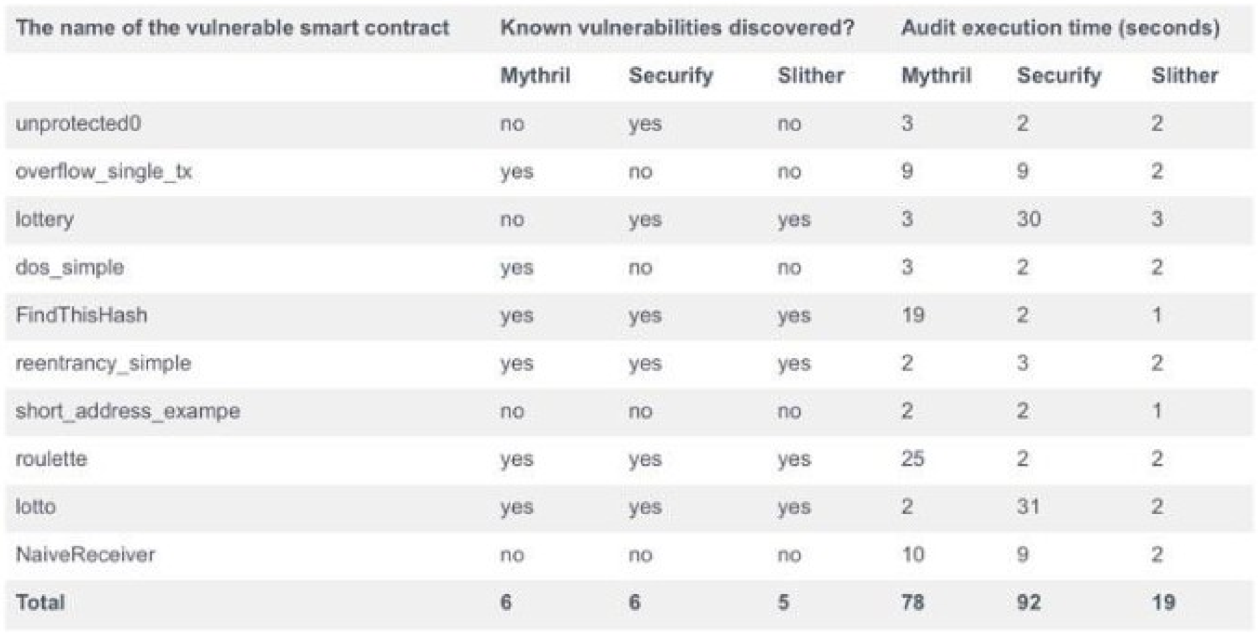
1. *Oyente* is a symbolic execution tool that works directly with Ethereum virtual machine (EVM) byte code without access to the high level representation. This mechanism is critical as the Ethereum blockchain only stores EVM byte code of contracts, not their source.
2. *Securify* is a scalable static analyzer tool for Ethereum Solidity contracts. Securify provides info on risk levels, vulnerability message types, smart contract name, and potentially vulnerable code.
3. *Mythril* is based on a combination of symbolic execution and concrete execution.

4. *Slither* is a source analyzer, leading in the field designed to provide rich information about Ethereum smart contracts.

Below are the results of an evaluation of smart contract static analysis tools through SolidiFI. It can be seen that every tool that has been used is unable to detect every bug. It's not actually clear which tool is leading. We could say that the overall efficiency is in decent levels, but in need of improvement.



The name of the secure smart contract	Number of false positives			Audit execution time (seconds)		
	Mythril	Securify	Slither	Mythril	Securify	Slither
BAToken	2	4	3	5	112	2
BlockWRKICO	3	0	2	27	9	2
EternalStorageProxy	0	3	2	1	30	1
FunctionXToken	2	2	2	5	2	1
HBToken	0	5	1	16	118	1
LayerX	2	0	3	2	2	2
NPXSToken	2	10	1	3	316	3
sHakka	0	0	1	2	2	2
ThePoolz	1	1	3	5	5	1
HumanStandardToken	1	0	2	2	3	2
Total	**13**	**25**	**20**	**68**	**599**	**17**



The name of the vulnerable smart contract	Known vulnerabilities discovered?			Audit execution time (seconds)		
	Mythril	Securify	Slither	Mythril	Securify	Slither
unprotected0	no	yes	no	3	2	2
overflow_single_tx	yes	no	no	9	9	2
lottery	no	yes	yes	3	30	3
dos_simple	yes	no	no	3	2	2
FindThisHash	yes	yes	yes	19	2	1
reentrancy_simple	yes	yes	yes	2	3	2
short_address_exampe	no	no	no	2	2	1
roulette	yes	yes	yes	25	2	2
lotto	yes	yes	yes	2	31	2
NaiveReceiver	no	no	no	10	9	2
Total	**6**	**6**	**5**	**78**	**92**	**19**

Figure 6: Results of the audit of vulnerable smart contracts.

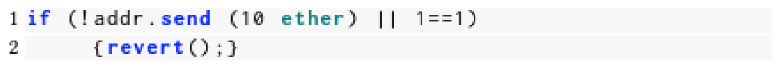
The Securify tool gives good results in a relatively short time. It has the most templates for different types of vulnerabilities and can detect simple logic errors, but cannot optimise gas costs. Securify reported 1-10 false positives in 6 out of 10 secure contracts. The Mythril tool reported 1-3 false positives in 7 out of 10 secure contracts. For vulnerable contracts, Mythril and Securify identified problems in 6 out of 10 contracts and Slither in 5 out of 10.

3.3 FLAWS

The question remains the same: Why some bugs are not detected and what are each tools' detection flaws?

Oyente was not able to detect many instanced of injected reentrancy, timestamp depencedncy, unhandled exceptions, integer overflow ' under flow, and TOD bugs. Oyente works on detection only re entrance bugs that are based on the use of (call.value). (Source: Luau et al. 2016). In the code of Figure 1, if two transactions are executed in one block and the attacker tires to change the order of the received transactions, error occurs. Oyente' s efficiency to detect all bugs successfully comes with a paradox due to the incompleteness of symbolic execution.

Mythril was the tool with the largest set of undetected bugs. Failed to detect many instanced of reentrancy , timestamp dependency, unchecked send, unhandled exceptions, integer overflow/under flow and use of tax.origin.



```
1 if (!addr.send (10 ether) || 1==1)
2       {revert();}
```

Figure 7: Unhandled exception code snippet 2.

In the line 1. “1==1” this is always gonna be true for Mythril. Therefore, the execution of the contract would be reverted in all cases by the function revert(), regardless of whether the send succeeds. This is incorrect as the execution should only be reverted on the fails of send. However, Mythril does not detect this as it only evaluates the send() part in the condition of the if-statement rather than evaluating the whole condition with the OR part (|| 1==1). We need to underline that Mythril was also very slow in term of the time it takes to analyze contracts.

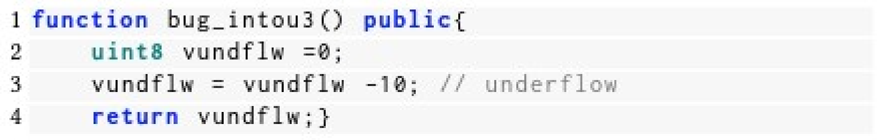
Securify was not able to detect several cases of injected bugs belonging to re-entrancy, unchecked send, unhandled exceptions, and TOD. In addition, there have been found many cases where Securify failed to analyze the injected contracts and threw an error.

4. DISCUSSION

Collecting all the available data regarding analysis tools and making good use of the gathered information, is not an easy work. There are plenty of articles and papers to be found and each one of them approaches the matter differently. To sum it up, most researches have concluded that the most popular vulnerability checked and detected by most of the static and dynamic analysis tools is ’re-entrancy’. The most popular analysis methodology employed by static analysis tools is symbolic execution and fuzz testing by dynamic analysis tools. Most of the tools have utilized static analysis, and some tools were found that employ a combination of static and dynamic analysis, i.e., hybrid analysis. It is concluded that hybrid analysis-based tools have considered more than 95% of security flaws. Generally, no security tool is thorough enough to replace an audit. It was also found that most of the tools’ source code is not openly accessible for evaluation purposes, which is a method that should be avoided since it is important to build trust among research community.

5. CONCLUSION

The most important finding of all the researches so far is that today’s popular smart contract auditing tools are quite weak at finding vulnerabilities in code. They keep missing up to half of the simple, typical vulnerabilities, not to mention the more complex ones. The analysed smart contract analysis tools produce different types of erroneous results and find different problems in the same set of smart contracts. The basic and the additional functionality of these tools varies significantly. We cannot be sure that any one of the above-mentioned tools is significantly better than the others for practical smart contract security analysis. Several general-purpose and highly specialised tools must be used simultaneously to improve the effectiveness of the analysis. Several approaches and tools have been developed that statically find security bugs in smart contracts [Feist, 2019; Luu, 2016; Mueller, 2018; Tikhomirov, 2018; Tsankov, 2018]. However, despite the prevalence of these static analysis tools, security bugs abound in smart contracts [Perez and Livshits 2019]. This calls into question the efficiency of the tools and the bug injection methods. As far as we know in the present time, there is no systematic method to evaluate static analysis tools for smart contracts regarding their effectiveness in finding security bugs.



```
function bug_intou3() public{
    uint8 vundflw =0;
    vundflw = vundflw -10; // underflow
    return vundflw;}
```

Figure 7: Undetected integer underflow bug

6. SOURCES

- Ethereum Smart Contract Analysis Tools: A Systematic Review
- https://www.researchgate.net › publication › 360128366_...Aug 29, 2022
- https://github.com/crytic/echidna
- https://ieeexplore.ieee.org/document/9762279
- https://arxiv.org/abs/2101.02377
- https://dl.acm.org/doi/abs/10.1145/3457337.3457841
- https://www.semanticscholar.org/paper/MuSC%3A-A-Tool-for-Mutation-Testing-of-Ethereum-Smart-Li-Wu/79cc88510f88d7a720f49fe939a51728ba75ff77
- https://www.euromoney.com/learning/blockchain-explained/what-is-blockchain
- https://www.ibm.com/topics/what-is-blockchain
- https://medium.com/coinmonks/smart-contracts-common-vulnerabilities-solidity-e64c5506b7f4
- https://blaize.tech/article-type/9-most-common-smart-contract-vulnerabilities-found-by-blaize/
- Etherscan. Etherscan. https://etherscan.io
- https://dl.acm.org/doi/10.1145/3395363.3397385, “How effective are smart contract analysis tools?” A. Ghaleb, K. Pattabiraman, 2020
- “A Survey of Tools for Analyzing Ethereum Smart Contracts”, G. Salzer, M. di Angelo
- New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE2018âĂŞ10299). https://medium.com/@peckshield/alert-new-batchoverflowbug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536, 2018
- solidity-security-blog. https://github.com/sigp/solidity-security-blog

- History of Ethereum Security Vulnerabilities, Hacks, and Their Fixes. https://applicature.com/blog/blockchain-technology/history-of-ethereumsecurity-vulnerabilities-hacks-and-their-fixes