Comparison of static analysis tooling for smart contracts on the EVM

Rick Fontein
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.l.h.fontein@student.utwente.nl

ABSTRACT

The interest in smart contracts running on a blockchain has increased lately. Since smart contracts live in a particularly hostile environment, they should be under rigorous scrutiny before deployment. Formal verification is desired, but often hard and time-consuming. Static analysis tools can help detect common mistakes and errors. Recently a set of static analysis tools have been developed specifically targeting the smart contract platform of the Ethereum project. This study aims to compare a set of these tools. A comparison study of static analysis tools helps inform smart contract developers of what these tool's capabilities. In addition, the results of this study highlight the features and usability of the tools, and suggest improvements.

Keywords

Blockchain, Smart Contracts, Ethereum, EVM, Static Analysis

1. INTRODUCTION

Blockchain has proven itself in the past few years with the widespread use and growth of Bitcoin. According to coinmarketcap.com, the cryptocurrency sees over 5 million USD traded in volume daily [2]. Bitcoin as a cryptocurrency is the original intention of the use of blockchain: a peer-to-peer network capable of making payments without a trusted party [18]. This is done with a shared public ledger and a protocol to reach consensus over said ledger. More recently, it has seen the adoption in areas broader than just a simple ledger. One notable new use is that of smart contracts, coined in 1996 by Nick Szabo, but popularized by the public blockchain Ethereum [20].

A smart contract is a piece of code that is stored in the blockchain. The nodes in the network execute the code and enforce the result of the execution with a consensus protocol. Since the code is run publicly, and more importantly distributed, smart contracts create a platform where applications that depend on fairness can thrive. Examples of such applications are sub-cryptocurrencies, multi-signature wallets, as well as other less financially oriented applications such as escrow or wills.

Smart contracts live in a particularly hostile environment. As described by Everts and Muller, once deployed they are unchangeable, autonomous, unstoppable, publicly visible and analyzable [14]. There have been incidents involving bugs in smart contracts that have led to considerable financial losses. Most notably the DAO incident in June 2016, where a smart contract containing the equivalent of 60 million USD was compromised [13].

Recently, there has been an effort on multiple fronts to help prevent such incidents from happening. Formal verification of smart contract code is desirable but often hard and time-consuming. Static analysis of the contract code is a lightweight approach that can help detect common mistakes and errors in smart contract programming [17]. A few tools performing such analysis have been developed over the past year. A broad comparison focused on tool features and relation to formal semantics has been done by the authors of the KEVM framework [15].

However, no in-depth comparison of static analysis tools has been performed. Hence this paper presents a comparison between three tools that analyze Ethereum smart contracts. Evaluation is useful on multiple fronts: It gives smart contract developers insight into these tools, what they feature and lack, how usable the tool is in terms of installation and analysis, how useful the tool's output is. It will also generate feedback on these relatively new tools. This feedback includes suggestions for usability improvements, expanding the detected mistakes and increasing the feature set.

The research is executed in three parts. First, an overview of the common mistakes made on the Ethereum platform is presented in section 3. Five mistakes that are particular to the Ethereum platform are discussed in detail. They provide a background of the Ethereum particularities involved, and how these can be abused. For the sake of brevity, mistakes that plague most programming languages and best practices of the Ethereum platform are only mentioned. Secondly, a benchmark of minimal sources containing the described mistakes is created based on the mistakes researched in the first part. The benchmark is then applied to the three tools, and the results are reported. Finally, a discussion of the results and recommendations is documented.

The contributions of this paper are:

- Overview of common mistakes specific to the Ethereum platform.
- Overview of the claimed and successfully detected mistakes by Oyente, Mythril and Porosity.
- Recommendations for improvement of the tool's output and feature set.

27^{th} *Twente Student Conference on IT* February 2^{nd}, 2018, Enschede, The Netherlands.

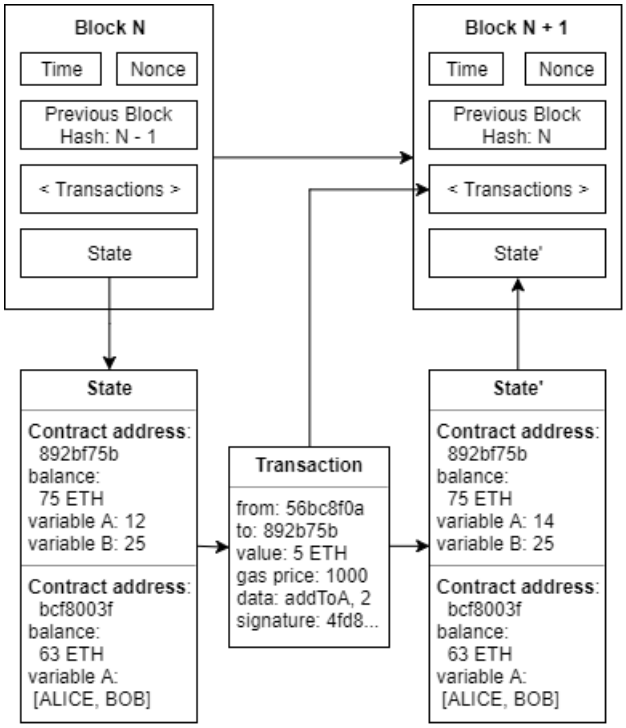

Figure 1. Example execution of a smart contracts on Ethereum. A transaction uses the state stored in the previous block to compute a new state'. Both the transaction and the new state' are stored in the new block.

- Recommendations for smart contract developers which tools to select.

2. BACKGROUND

2.1 Blockchain

A blockchain consists of lists of transactions, called blocks, which are linked together cryptographically. Blocks typically contain a hash that serves as a pointer to the previous block, a timestamp and a list of transactions. However, other data can be stored in the blocks as well.

Bitcoin uses the blockchain as a distributed ledger in a peer to peer network. The peers adhere to a protocol for adding and validating new blocks to the chain. Once a new block is added, the data in it can not be altered without altering the blocks succeeding it.

In public platforms like Bitcoin or Ethereum, consensus over the blocks is achieved by a Proof-of-Work scheme. The peers have to solve a cryptographic puzzle by brute-forcing it. The peer that finds the solution submits a new block containing the solution and transactions to the chain. If the block is valid, it is added to the chain and the peer receives a reward for finding the solution.

2.2 Smart contracts

Smart contracts are 'Autonomous agents' that are stored on the blockchain. In Ethereum's case, smart contracts have an address, Ether balance (the native cryptocurrency of Ethereum), code in the form of EVM byte-code instructions and a state of the defined variables. The contract's code can be executed by sending a transaction to the contract's address. This transaction includes some Ether to serve as an execution fee, and the necessary arguments to execute the code defined in the contract. The transaction computes a new state based on a previous state stored in the preceding block. Finally, the transaction and the new state are stored in a new block. An illustration of a transaction can be found in figure 1.

2.2.1 *Ethereum Virtual Machine*

Smart contracts on the Ethereum platform are executed in the Ethereum Virtual Machine (EVM). The EVM is a limited low-level stack machine capable of sequential execution of its corresponding byte-code. EVM byte-code consists of a set of opcodes with arguments, these instructions are specific to Ethereum and can, therefore, provide access to variables of the environment, such as block numbers and hashes.

Execution of a transaction on the EVM is not without cost. Every transaction to a contract contains Ether, the native Ethereum currency, and a 'gas price'. Every EVM instruction costs a certain amount of gas. The total execution gas of a transaction multiplied by the gas price is the execution fee. This fee is deducted from the Ether that was sent with the transaction; the rest is returned to the sender.

2.2.2 *Solidity*

The EVM byte-code is often compiled from higher level languages, most notably Solidity, a JavaScript-like language. An example of Solidity can be found in listing 1. Like many programming languages, it offers common features like control flow structures, types (booleans, integers, arrays maps), functions, enums, and structs. Specific to Solidity is a set of EVM related globally available variables and functions. The most important and their purpose are highlighted briefly in table 1.

Variable	Meaning
block.number	Current block number
block.timestamp	Current block time
msg.sender	Sender of the current call
msg.gas	Remaining gas allowance
msg.value	Ether sent with transaction
tx.gasprice	Gas price of the transaction
tx.origin	Sender of original transaction (full callchain)
Function	**Meaning**
throw	Abort transaction, revert to old state
<address>.send()	Sends Ether to address, returns false on failure
<address>.transfer()	Send Ether to address, throws on failure
<address>.call()	Calls function at address with Ether

Table 1. Ethereum specific variables and functions.

3. COMMON MISTAKES IN EVM

We present an overview of some common mistakes and errors in smart contract programming that a malicious user or miner, might exploit. Of all the errors featured on Ethereum safety and best practices lists [11, 10], we discuss five that are specific to the Ethereum platform in more detail. Some of these errors are infamous for Ether theft. For the sake of brevity, mistakes that plague most programming languages and best practices are only mentioned in section 3.6.

```
contract Storage{
 mapping(address => uint) balances;

 function donate(address to) payable {
  balances[to] += msg.value;
 }

 function balance(address who){
  return balances[who];
 }

 function withdraw(uint amount){
  //check if sufficient balance
  if(balances[msg.sender] >= amount) {

   //send funds, re-entrant vulnerable
   msg.sender.call.value(amount)());

   //update balance sheet
   balances[msg.sender] -= amount;
  }
 }
}
```

Listing 1. Storage contract. Allows users to store Ether in this contract. The `withdraw` function is vulnerable to re-entrancy.

```
Contract Attacker{

 //location of vulnerable contract
 Address storage;

 function attack(){
  //Initial call to withdraw to start
      attack
  storage.withdraw(1);
 }

 //fallback, executed when no function
     is specified
 function () payable {

 //check if storage still has Ether
  if(storage.balance >= msg.value) {
   //re-entrant call to storage
   storage.withdraw(1);
  }
 }
}
```

Listing 2. Attacker Contract. An example contract that performs a re-entrant attack on the storage contract in listing 1.

3.1 Re-entrancy

A function is called re-entrant when it can be interrupted in the middle of execution and called again before completing the interrupted execution. Re-entrancy is the main culprit that depleted the infamous DAO contract, causing a hard fork in the Ethereum chain [13]. The bug relies on two particularities of the EVM. `call()` and other low-level EVM instructions that forward Ether hand over program execution to the callee. When the destination address is a wallet, it simply deposits the funds, returns and resumes execution. Secondly, the EVM allows for fallback functions. These are nameless functions in a contract that are executed whenever no function name is specified in a call to the contract. When the destination of a `call()` is a contract, the fallback assumes control of the program execution. From there, the fallback can issue re-entrant calls.

Consider the Storage contract in listing 1. It has an internal sheet for keeping track of the balances, a function to donate to an address, checking balances and withdraw funds from the contract. This contract is vulnerable in the `withdraw()` function for re-entrancy: `call()` in this function sends Ether to the callee, it forwards any remaining gas it has. The attack contract in listing 2 abuses the `call()`: When the withdraw function of the storage sends a `call()` to the attacker, the attacker takes control of the program flow in the attacking contract's fallback function. Before the Storage contract has a chance to update the internal balance sheet, the attacker recursively calls `withdraw()` again. The check for sufficient funds will succeed because the internal sheet has not been updated yet, and the requested withdrawal funds will be sent again, depleting the Ether balance of the storage contract.

3.2 Timestamp dependency and weak random generators

Contracts functions that rely on block or network specific values can be predicted and abused by miners. Random number generation (RNG) plays an important role in computer science. In a public deterministic environment such as Ethereum, it is particularly hard to generate random numbers. Naive solutions use values that are time specific to either the network or the block containing the transaction. Examples of such values are timestamps, coinbase (the address of the miner), block numbers, gas limits or block difficulty. Results of functions that rely on such values can be predicted by a malicious miner, and consequently, supply input that is advantageous to an attacker. A notable example of a production contract vulnerable is theRun, a gambling contract that uses timestamps to generate a random number used to award a jackpot [8].

3.3 Transaction order dependency

Smart contracts that are called based on a perceived state of said contract should take transaction ordering into account. There is a time delay between the transaction commitment (submitting to the network) and its confirmation and inclusion in a block. Consider a contract that implements a puzzle. It has two functions, submit solution and update reward. Upon submitting the correct answer a reward in Ether is won. The owner of the contract could listen to the network and upon detection of a transaction that solves the puzzle, quickly send a transaction that lowers the reward. Since no transaction is included in a block yet, the reward paid out is at the mercy of the miner which determines which transaction is executed first. The owner has a certain chance of beating the solving transaction. Naive implementations of auctions and (stock) exchanges are often vulnerable to this issue.

3.4 Call stack depth and checks on return values

Calls to other contracts that contain Ether can fail, continuing execution based on the assumption of a successful call could have a significant impact. Consider the king contract in listing 3: it is a simple public display of wealth. Users send Ether to the contract, if they send more Ether than the current king, they are crowned king and the old

Mistake	Oyente		Mythril		Porosity	
	Claimed	Detected	Claimed	Detected	Claimed	Detected
Re-entrancy	✓	✓	✓	✓	?[1]	✓
Timestamp dependence	✓		✓	✓	?[1]	
Call depth	✓	✓	?[2]	✓	?[1]	
Transaction order dependence	✓[3]	✓	✓			
Check on CALL return value	?[2]	✓	✓	✓		
Integer under/overflow			✓	✓		
Predictable RNG			✓	✓		
Use of tx.origin			✓	✓		
Call to untrusted contract			✓	✓		
Unprotected Functions			✓	✓		

[1] None of these are explicitly mentioned as detectable by porosity
[2] Missing check on call return value is part of call stack depth exploit
[3] Tool output labels it as 'money concurrency'

Table 2. Claimed detection and experiment results

king's Ether is returned. The `send()` instruction transfers the funds back to the old king. If the `send()` fails, program execution is still continued and a new king is crowned. When no other functions that allow withdrawal of Ether from the contract are available, the old king's funds will be trapped indefinitely.

An attack vector that abuses the call stack limits of the EVM exists. The stack of the EVM is limited to 1024 frames. A new frame is created every time a function is called. If an attacker would purposefully call itself recursively 1023 times before calling `claim()` (frame 1024), they will have used up the 1024 frames available. The `send()` instruction to return the old king's funds would create the 1025th frame and thus fail.

Protocol changes to the way gas is calculated introduced a practical limit to the number of frames that can be created. As a result a recursive call would run out of gas well before the hard limit of 1024 is reached [12]. Nevertheless, not checking the return values of calls or Ether transfers poses a significant security risk if another attack vector would present itself.

```
1  contract King{
2   address king;
3   uint wealth;
4
5   function claim() payable{
6    //more funds than current king?
7    if (msg.value > wealth){
8
9     //Return old king's Ether
10    //No check on return value (success),
          program execution will always
        continue.
11    king.send(wealth);
12
13    //Crown the new king
14    king = msg.sender;
15    wealth = msg.value;
16
17   } else {
18    //Not wealthier, reject transaction.
19    throw;
20   }
21  }
22 }
```

Listing 3. King Contract. The user that sends the most Ether is crowned king. The return value when sending funds to the old king is not checked, on failure, the funds are trapped in the contract.

3.5 Use of tx.origin

Usage of `tx.origin` for access restriction allows for impersonation attacks. All functions in a smart contract are callable by everyone. The address of the caller is often used for access restriction. `tx.origin` returns the address of the wallet that issued the first call for a transaction. `msg.sender` on the other hand returns the address of the wallet or preceding contract that called the function. Consider the following call chain: Wallet A → Contract B → Contract C. Then, in contract C: `tx.origin` is the address of wallet A and `msg.sender` is the address of contract B.

If an attacker would succeed in having a user (Wallet A) interact with a seemingly unrelated contract (Contract B), the unrelated contract could issue a call to the vulnerable contract (Contract C). Since `tx.origin` is the address of the user, the attacker could potentially execute access restricted code in the vulnerable contract.

3.6 Other mistakes and best practices

Other general mistakes such as integer over- and underflow exist within Solidity. Wrong business logic or logic that fails to check for such events could be exploitable.

Mismatching the exact name of the contract and constructor is one such issue and often easily missed. (mismatched capitalization, British American spelling differences, etc.) The intended constructor compiles to a normal function instead of a one-time executable constructor. If it contains code to set the owner or other critical access values, this code will become publicly callable.

Finally, the Ethereum community has compiled lists of best practices [11, 10]. The list contains Ethereum specific code styles, common business logic mistakes and smart contract tips for implementing update mechanisms and fail-safes.

4. TOOLS

In this section we list three tool that we compare. For every tool we provide some background, why this tool is in this comparison and the tool features including what mistakes they claim to detect.

Furthermore we compare the documentation of each tool on the availability and completeness of installation instructions, usage instructions, implementation details and feature documentation.

Finally, the tool installation process and installation time is compared.

An overview of the last two parts is provided in table 3 on page 6.

4.1 Oyente

Oyente is a static analysis tool based on symbolic execution. It has been developed in early 2016 as part of a broader research paper on smart contract security [17].

Symbolic execution is an analyses technique that aims to determine which inputs cause which program branches to execute. It does this by defining the input as a symbolic value, and expressing each conditional path in this symbolic value. This value can then be used to calculate what input values trigger that specific path [16]. Oyente implements a symbolic executor that faithfully emulates the EVM's instruction set. It uses Microsoft Research's Z3 Theorem prover to decide satisfiability of the explored paths.

Oyente is the only tool of the three discussed here that has performed a benchmark on their own tool. The authors scraped the Ethereum network's first 1,459,999 blocks yielding 19,366 contracts. A total analysis time of roughly 3000 hours on a large Amazon EC2 cluster is reported. The flagged contracts that had their source code in Solidity available were manually inspected to determine true and false positives.

Oyente's main goal is static analysis of smart contracts, it makes a solid first candidate in this comparison.

4.1.1 Features

- Analysis with symbolic execution
- Input can be supplied as EVM byte-code and Solidity source code.
- Input can be supplied locally and via URL's.
- Assertion checker for contract assertions. Assertions are true-false expressions which are assumed to be always true at runtime, if it fails at runtime, the transaction is rejected.

Table 2 on page 4 provides an overview of the claimed detected errors per tool. Some particularities in naming exist: The paper appropriately names the transaction order dependency as such, but the tool outputs this as 'Money concurrency bug.' Furthermore, in the documentation of the call-stack error implementation it is explained that detection uses a check of the return value of call(), covering the broader category of errors of which the call stack bug is part of: explicit check on call return value.

4.1.2 Documentation

The tool's main documentation is the accompanying paper. The paper details design details including Oyente's architecture as well as an implementation overview. The responsibilities of the four main source files are described. The section that discusses the core analysis portion of the code base explains how the 4 errors that the paper focuses on are detected. A limited pointer to extend the analysis capabilities of the tool is provided as well. Furthermore, the repository's readme file contains an installation and quick start guide. The file code.md reiterates the code structure as described in the paper [6].

4.1.3 Installation

Oyente is written in python 2.7. It provides three ways of installing the tool: A prepared docker container ready to go, installation from the python repository 'pip' and step-by-step instruction for manual installation of dependencies and the tool. Installation of Oyente via pip took 3 minutes on Ubuntu 16.04/Intel I7-3630QM/8GB RAM. Assuming developer tools that are essential to the platform (Solidity compiler, web3) are already installed, most time was spent pulling and installing the dependencies specified in the pip package.

4.2 Mythril

Mythril is a tool whose main goal was chain exploration, over time it has added a module for static analysis. Tools like Etherscan, Remix and testrpc existed before Mythril to browse, (de)compile, disassemble and debug contracts. However, certain tasks such as searching the chain were not possible with these tools. Mythril aimed to fill that gap by adding a comprehensive blockchain exploration tool, as well as an Ethereum disassembler and Ethereum scripting abilities in Python. Python scripting allowed for an concolic testing module named 'Laser' to be added to the tool.

Concolic testing is a hybrid analysis technique that performs symbolic execution along a concrete execution path [19]. Like Oyente, Microsoft Research's Z3 Theorem prover is used in conjunction with the symbolic execution engine laser-ethereum present in Mythril.

4.2.1 Features

- Analysis with concolic testing.
- Input can be supplied as EVM byte-code and Solidity source code.
- Input can be supplied locally and remote addresses of on chain contracts.
- Control Flow Graph (CFG) visualization. A CFG is a representation of a programs possible execution paths in graph notation.
- Blockchain exploration, search options allowing simple boolean expressions:
  - Specific contracts
  - Specific function calls
  - EVM opcode sequences
- Disassembler for byte-code string or address of an on-chain contract.
- Cross referencing of contracts. Useful for finding contracts that reference other contracts.

Table 2 on page 4 provides an overview of the claimed detected errors per tool.

4.2.2 Documentation

The main documentation of Mythril is the Readme of the Github repository [5]. It contains an installation section, as well as usage guides for security analysis, CFG output, blockchain exploration and provided utilities (disassembler and cross-references). No implementation details are published apart from a generated PyDoc based on limited comments in the source code of the analysis modules.

4.2.3 Installation

Mythril is built on Python 3.5. It provides two ways of installing: Using package manager pip to install the tool and its dependencies or downloading the Github repository and running an included installer. Relatively to Oyente, installation is slow: Mythril has a larger set of dependencies it needs to pull and build. The pip install took 11 minutes on Ubuntu 16.04/Intel I7-3630QM/8GB RAM. Linux builds with newer versions of certain libraries (e.g.: LibSSL) like Ubuntu 17.10 have dependency issues with Ethereum libraries, which at the time of writing, fails Mythril from building.

4.3 Porosity

Porosity is a decompilation tool that tries to translate EVM bytecode into Solidity. Porosity was an effort from the side of reverse-engineers in their desire to having access to source code. As mentioned on the Github page, this would then allow for static and dynamic analysis of compiled contracts, including vulnerability discovery [6].

4.3.1 Features

- Analysis of contracts, the documentation does not detail how this is achieved.
- Input can be supplied in the form of EVM byte-code.
- ABI parser: Lists the functions in the contract from the pre-loader EVM byte-code.
- Disassembler of EVM byte-code.
- CFG vizualisation.
- Decompilation of EVM byte-code into Solidity.

Table 2 on page 4 provides an overview of the claimed detected errors per tool. The tool's Github pages mentions the ability of vulnerability analysis within the tool, as well as an example tool output containing a warning for a possible re-entrancy vulnerability. The white paper lists the re-entrancy, call stack depth and timestamp dependency mistakes, but makes no explicit claims that Porosity can detect any of these.

4.3.2 Documentation

Documentation for Porosity is limited compared to Oyente and Mythril. The Github repository provides a readme with the steps and execution of an example decompilation. The white paper that is contained in the Github repository has little documentation of the tool itself, it merely explains the workings of Ethereum, and the same decompilation example as in the readme. The wiki page on Github contains an exact copy of the text in the paper.

4.3.3 Installation

The readme file has an overview of the three major platforms: Windows, Linux and Mac OS X. All indicating a build success or supported status. The Github provides binaries for windows only. The existence of a makefile in the repository is the only indicator on how to install on a platform other than windows. No dependencies are listed anywhere other than calls in the actual source code of Porosity. Building Porosity with the makefile will cause it to fail on systems that do not have the correct dependencies. A reiterative process of installing or upgrading the missing libraries follows until the tool fully compiles. 26 minutes were spent form the first build attempt to a successful build.

Documentation	Oyente	Mythril	Oyente
Detected Errors	✓	✓	?[1]
Implementation	✓	✗	✗
Dependency list	✓	✓	✗
Usage instructions	✓	✓	✓
Installation instructions	✓	✓	✗[2]
Installation time (minutes)	3	11	26
Last updated	2018-1-5	2018-1-16	2018-1-16

[1] No explicit claims, three mistakes are discussed in documentation.
[2] Windows compiled binaries are provided.

Table 3. Documentation and installation

5. EXPERIMENTS

We apply a benchmark of Solidity sources on Oyente, Mythril and Porosity, the tool output is then analyzed on two aspects. First we analyze the tool output for the errors they report on the different sources. The benchmark consists of concise smart contracts that are vulnerable to the mistakes described in section 3. The results are then compiled in table 4 on page 7. This table contains for every source in the benchmark the manually confirmed mistakes, and the results of the static analysis of the tools.

Secondly, the tool output itself is analyzed for several factors: How the mistake is reported. If and how the tool pinpoints the error in the source. If the tool provides context and finally if it provides suggestions for improvement.

The benchmark can be extended by providing sources and repeating the process described above. A fourth tool can be added to the comparison in a similar fashion.

5.1 Results

Table 2 on page 4 provides an overview of the detected errors. The results of the specific sources of the benchmark have been recorded in table 4 on page 7. Particularities and oddities of the results of certain results are noted in the footnotes.

5.2 Detected mistakes

The version of Oyente used is different from the one used by Luu et al [17]. The Ethereum community has developed it further. The community version of Oyente detects three out of the four claimed mistakes. The fact that it fails to recognize timestamp dependency is surprising, since the bug and a quantitative benchmark is discussed by Luu et al. The production on-chain source contract that was used to illustrate the mistake in the paper[8] has no flagged vulnerabilities either.

Mythril detects most of the claimed errors. It does not detect the transaction order vulnerability present in the puzzle contract. Oyente correctly reports this error and identifies the two separate flows leading to the vulnerability. The documentation of Mythril claims detection, but does not provide background information, implementation details or a link to the source.

Porosity did not make explicit claims which errors it detects. Our benchmark shows that it only detects re-entrancy, not the other two that were discussed in the documentation. Moreover, the tool's decompilation ability seems to be very limited. One of our sources timed out after 13 minutes, two sources segfault before completing decompilation. Even the decompilation of the storage contract,

Contract	Confirmed mistakes	Oyente	Mythril	Porosity
storage	re-entrancy	✓	✓	✓
king	call stack depth/Check on return value	✓	✓[1]	✗
roulette	timestamp dependence	✗	✓[1,2]	✗[3]
puzzle	transaction order dependence	✓	✗	✗[4]
underflow	integer underflow	✗	✓	✗
overflow	integer overflow	✗	✓	✗
origin	usage of tx.origin	✗	✓	✗[3]
weak random	predictable RNG	✗	✓[1]	✗
ether send	unprotected functions	✗	✓	✗

[1] Additional error: taintable critical storage; The destination address of a call forwarding Ether can be set directly or via taintable storage.
[2] Additional error: Integer overflow. Intended behaviour.
[3] Segfault in execution, failed to decompile.
[4] Execution halted after 13 minute timeout.

Table 4. Benchmark results

which correctly identified re-entrancy, is incorrect. The output reports two re-entrancy calls and highlights two lines of decompiled code. The lines contain exactly the same call, including parameter. The first of these calls is supposed to be a throw statement.

It should be noted that this benchmark will not provide a definitive answer as to whether or not the tool is capable of detecting a specific type of mistake. Quantitative analysis is needed to answer that question as well as to determine the false and true positive rates. It does provide a basic insight in the tools capabilities as the sources are slimmed down and purposefully programmed to demonstrate these mistakes.

5.3 Tool output

Oyente claims detection for 4 errors, the tool output displays a boolean flag for each of these. If a true flag is set and a Solidity source was supplied, it outputs the line and character number where the mistake was detected. Underneath it prints the specific instruction that caused the flag. For the transaction dependency, it identifies the multiple flows that caused the flag. These flows contain the same line, character and instruction information. No contextual information or suggestion for improvements are provided.

Mythril shows the function name and the PC address where the mistake is located. Along with the location it provides a brief description of the mistake and how it affects the source. If a taintable storage location is found, it lists which functions set or alter values of the specific storage slot.

Porosity attempts to decompile the supplied EVM code. If a mistake was detected, the tool highlights the decompiled line that is affected in red, along with it an error message of the possible vulnerability. Furthermore, the implementation of supported EVM instructions is limited. Every source in our benchmark produced a warning of unsupported instructions, causing incomplete or faulty decompilations. Like Oyente, no contextual information or suggestion for improvements is provided.

6. RECOMMENDATIONS

6.1 Smart Contract Developers

Judging the tools based on the results in this research, Mythril has the largest set of mistakes it can detect. Although it should be noted that none of the tools in this paper can successfully detect all the mistakes discussed in section 3. Mythril is therefore the best candidate for developers seeking static analysis. This is aided by the brief contextual messages in Mythril's output, making it more accessible for those less informed. For more complete coverage, running Oyente to detect transaction order dependencies is a good solution. The limited decompilation ability and the limited amount of mistakes Porosity can detect, make it the least appealing of the three.

6.2 Tool Developers

Oyente can be improved by fixing the timestamp dependency detection. In the same spirit, it could add detection for dependency of predictable block properties such as block difficulty. Moreover, Oyente uses symbolic execution, a by product of this process is a CFG. As mentioned in the Oyente repository, with some effort, an export of this CFG could be made available for manual analysis.

Support for transaction order dependence can still be added to Mythril. Implementation details are limited to a generated PyDoc, this could be improved upon.

Porosity analysis is limited, before extending vulnerability detection, the supported EVM instruction set should be expanded for complete decompilation. In addition, Porosity is not stable. Several sources in our benchmark failed to decompile due to segfaults or timeouts. Furthermore, the documentation is incomplete. A listing of what mistakes it can detect should be included in the documentation. Installation instructions for platforms other than Windows can be added as well.

7. FURTHER RESEARCH

As mentioned in the discussion of the benchmark results, quantitative analysis on the tool can give a more definitive answer of the tool's capabilities of detecting errors, including providing statistics of true and false positive rates. The benchmark used in this paper are slimmed down contracts containing the mistakes mentioned in section 3. A benchmark containing larger, more complex (more realistic)

Mythril and Oyente use symbolic execution: they compute a symbolic value for all program branches. More complex sources can take exponentially more analysis time, a comparison based on time usage is therefore useful.

More code analyzers for the Ethereum platform can be added to the comparison. Dr. Y's [3] is another symbolic executor written in OCaml. Manticore has recently added support for Ethereum, another symbolic execution tool [4]. Securify is a tool that claims formal analysis and strives

to achieve no false negatives, implementation is not public [7].

Other areas of interest are those of different programming Paradigms. Solidity is one of the higher level languages compiling to EVM byte-code. Other higher level languages are available: Vyper[9] and Bamboo[1] are security focused by design. As a result these languages are less flexible compared to Solidity. Vyper for example has no support for unlimited loops.

8. REFERENCES

[1] Bamboo github repository. `https://github.com/pirapira/bamboo`. Accessed: 2018-01-21.

[2] Cryptocurrency market capitalizations. `https://coinmarketcap.com/`. Accessed: 2017-12-01.

[3] Dr. y's ethereum contract analyzer. `http://dry.yoichihirai.com/`. Accessed: 2018-01-21.

[4] Manticore github repository. `https://github.com/trailofbits/manticore`. Accessed: 2018-01-21.

[5] Mythril github repository. `https://github.com/b-mueller/mythril`. Accessed: 2017-12-01.

[6] Porosity github repository. `https://github.com/comaeio/porosity`. Accessed: 2017-12-01.

[7] Securify: Formal verification of ethereum smart contracts. `http://securify.ch/`. Accessed: 2018-01-21.

[8] 'the run' smart contract. `https://etherscan.io/address/0xcac337492149bdb66b088bf5914bedfbf78ccc18`. Accessed: 2018-01-17.

[9] Vyper github repository. `https://github.com/ethereum/vyper`. Accessed: 2018-01-21.

[10] Ethereum smart contract security best practices. `https://consensys.github.io/smart-contract-best-practices/`, Jun 2016. Accessed: 2018-01-17.

[11] Ethereum contract security techniques and tips. `https://github.com/ethereum/wiki/wiki/Safety`, Dec 2017. Accessed: 2018-01-17.

[12] V. Butarin. Long-term gas cost changes for io-heavy operations to mitigate transaction spam attacks. `https://github.com/ethereum/eips/issues/150`, Sep 2016. Accessed: 2018-01-16.

[13] V. Buterin. Critical update re: Dao vulnerability. `https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/`, Jun 2016. Accessed: 2017-12-01.

[14] M. Everts and F. Muller. Will that smart contract really do what you expect it to do?, 2017.

[15] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Dayan, D. Guth, and G. Rosu. *KEVM: A Complete Semantics of the Ethereum Virtual Machine*. Aug 2017.

[16] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. cited By 1257.

[17] 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.

[18] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2009.

[19] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[20] N. Szabo. Smart contracts: Building blocks for digital markets, 1996.