An Insecurity Study of Ethereum Smart Contracts

Bishwas C. Gupta, Nitesh Kumar, Anand Handa(✉), and Sandeep K. Shukla

C3i Center, Department of CSE, Indian Institute of Technology, Kanpur, Kanpur, India
{bishwas,niteshkr,ahanda,sandeeps}@cse.iitk.ac.in

Abstract. Ethereum is the second most valuable cryptocurrency, right after Bitcoin. The most distinguishing feature of Ethereum was the introduction of smart contracts which are essentially small computer programs that sit on top of the blockchain. They are written in programming languages like Solidity and are executed by the Ethereum Virtual Machine (EVM). Since these contracts are present on the blockchain itself, they become immutable as long as the blockchains integrity is not compromised. This makes it a nightmare for security researchers as the vulnerabilities found cannot be patched. Also, since Ethereum is a public blockchain, all the contract bytecodes are available publicly. The DAO and the Parity attack are two prominent attacks that have caused great monetary losses. There are many tools that have been developed to cope with these challenges. However, the lack of a benchmark to compare these tools, non-standard vulnerability naming conventions, etc. make the job of a security analyst very difficult.

This paper provides the first ever comprehensive comparison of smart contract vulnerability discovery tools which are available in the public domain based on a comprehensive benchmark developed here. The benchmark development is based on a novel taxonomy of smart contract vulnerabilities which has been created after a thorough study of security vulnerabilities present in smart contracts.

Keywords: Ethereum blockchain · Smart contracts · Security vulnerability discovery tools

1 Introduction

1.1 Ethereum

Ethereum is also a cryptocurrency backed blockchain like Bitcoin. It uses similar techniques like proof of work (it will eventually move to a proof of stake based consensus algorithm called Casper), hash pointers (Ethereum uses KECCAK-256), etc. However, the main difference between Ethereum and Bitcoin is that

Partially Supported by Office of National Cyber Security Coordinator (NCSC), Government of India.

L. Batina et al. (Eds.): SPACE 2020, LNCS 12586, pp. 188–207, 2020.
https://doi.org/10.1007/978-3-030-66626-2_10

unlike Bitcoin which is just a distributed ledger of transactions, Ethereum can also run small computer programs which allow developers to develop decentralized applications (or dApps). Also, unlike Bitcoin whose founder(s) are unknown, Ethereum is the vision of Vitalik Buterin, who wrote the white paper [9]. It is maintained by the Ethereum Foundation.

Unlike Bitcoin, Ethereum has two kinds of addresses [12] - **Externally Owned Accounts (EOAs)** which are owned through public-private key pairs and **Contract Accounts** these are special accounts which are controlled by the smart contract deployed on them. They can be triggered only by an EOA. Like Bitcoin, the users have to pay a small transaction fees for each transaction they want to be included in the blockchain. This is paid in Ethereum's native currency called Ether.

1.2 Smart Contracts

Smart Contracts are essentially small programs that exist on the blockchain and are executed by the Ethereum Virtual Machine (EVM). They do not need any centralised trusted authority like banks since all the functionality required is implemented in the smart contract logic, and since the code itself resides on the blockchain, we can be sure that it has not been tampered with. This property of being immutable is crucial in financial applications like escrow and other payments. Also, it allows developers to develop other smart applications by utilizing the power of blockchain technology. However, the concept of smart contracts is not new. It was introduced by Nick Szabo [46] in 1997.

1. **EVM:** EVM stands for Ethereum Virtual Machine which serves a similar purpose that Java Virtual Machine (JVM) does for Java by providing a layer of abstraction between the code and the machine. This also makes the code portable across machines. It also gives the developers an option to code in their smart contract language of choice, as finally all the programs written in different languages are translated by their respective compilers to EVM byte-code. The Ethereum Yellow Paper [51] explains the intricate workings of the Ethereum Virtual Machine in great detail.
2. **Smart Contract Programming:** The most popular programming language for Ethereum Smart Contracts is Solidity. It is a language similar to Javascript and C++, making it easy for existing software developers to write solidity code. Other languages, though not as popular are Vyper and Bamboo. Before Solidity was released, languages like Serpent and Mutan were used which have since been deprecated [29]. The compiler (solidity's compiler is called `solc`) converts the source code to EVM bytecode. This code is called the contract creation code. This is like a constructor to put the contract bytecode on the blockchain and can be executed by the EVM only once to put the run-time bytecode on the chain. The run-time bytecode is the code that is executed by the EVM on every call of the contract. The run-time bytecode also contains a swarm hash of the metadata file. This file can contain information like functions, compiler version, etc. However, this is still an experimental feature and not many have uploaded the metadata to the Swarm network [29].

3. **Ether and Gas:** Ether is the native cryptocurrency of the Ethereum network. Gas is another feature of Ethereum that separates it from Bitcoin. Since different smart contracts require varying amounts of computational power and time, it would be unfair to the miners to base the transaction fees just on the length of the transaction or have a constant transaction fees. Gas is a unit introduced by Ethereum that measures the computational work done. Each operation has an associated gas cost. However, gas is different from Ether as the value of the latter is market dependent but that does not change the 'computational power' required to execute the contract. Therefore, every transaction mentions a gas price which is the price a person is willing to pay in ether per unit of gas. The combination of these two give the transaction fees in ether.

1.3 Smart Contract Security

The biggest advantage of smart contracts - their immutability also poses the biggest threat from a security standpoint. This is because any bug found in the smart contract after deployment cannot be patched. Recent attacks like the DAO attack and the Parity attack have caused massive monetary losses. In such a scenario it becomes imperative to develop and interact with smart contracts that are secure. To achieve this goal, various tools have been developed by security researchers. However, the lack of organized information around smart contract security issues, a proper vulnerability naming convention and classification, and a benchmark to compare existing tools make the job of a security researcher very difficult.

The remainder of the paper is organized as follows - Sect. 2 introduces the various smart contract vulnerabilities. Section 3 describes the need for a new taxonomy and our novel taxonomy for smart contract vulnerabilities. In Sect. 4, we demonstrate our Vulnerability Benchmark for analyzing the security tools. Section 5 introduces the different security tools and highlights the results of the tools on the benchmark. Finally, we have past related works in Sect. 6 and concluding remarks in Sect. 7.

2 Smart Contract Security Vulnerabilities

We divide the security vulnerabilities in Ethereum into two broad categories - Blockchain 1.0 and Blockchain 2.0 vulnerabilities. Blockchain 1.0 vulnerabilities include security vulnerabilities that are present in most blockchain based systems and that Ethereum shares with its predecessors like Bitcoin. Blockchain 2.0 vulnerabilities include vulnerabilities introduced in the system because of the presence of smart contracts. Our work is concerned with the Blockchain 2.0 (Smart Contract) vulnerabilities. However, the Blockchain 1.0 vulnerabilities are introduced for completeness.

2.1 Blockchain 1.0 Vulnerabilities

1. **51% attack:** In proof of work, the miners try finding the nonce value to solve the given cryptographic puzzle. However, if miner(s) get control of more than 51% of the compute power in the network then they essentially control what goes into the blockchain - compromising its integrity [35].
2. **Double Spending:** Double spending [32] occurs when the attacker uses the same cryptocurrency more than once. This is done by leveraging race conditions, forks in the chain, or 51% attacks.
3. **Selfish Mining:** In selfish mining [23,41], a malicious miner does not publish the block immediately after solving the proof of work puzzle. Instead, reveals it only to its pool members which then work on the next block, while the other network continues working for essentially nothing [31].
4. **Eclipse Attack:** In eclipse attacks [28], the victim's incoming and outgoing connections are taken over by attacker (using a botnet or otherwise). This gives a filtered view of the blockchain to the victim.
5. **BGP Hijacking Attack:** The border gateway protocol (BGP) is used for handling routing information over the internet. BGP hijacking is a common attack. However in the context of public blockchains, it can be used to create unwanted delays in the network [34].
6. **Private key security:** Private keys are used to control the addresses in Ethereum. It is a security problem in itself to store these keys securely. As blockchain is a decentralised system, there is no way to report a stolen private key and prevent its misuse.

2.2 Blockchain 2.0 Vulnerabilities

1. **Re-entrancy:** A re-entrancy condition is when a malicious party can call a vulnerable function of the contract again before the previous call is completed: once or multiple times. This type of function is especially problematic in case of payable functions, as a vulnerable contract might be emptied by calling the payable function repeatedly. The `call()` function is especially vulnerable as it triggers code execution without setting a gas limit. To avoid re-entrancy bugs, it is recommended to use `transfer()` and `send()` as they limit the code execution to 2300 gas [40]. Also, it is advised to always do the required work (i.e. change the balances, etc.) before the external call. The DAO Attack is the most famous re-entrancy attack which lead to a loss of US$50 Million [13] and resulted in the chain being forked into two - Ethereum and Ethereum Classic.
2. **Authorization through** `tx.origin`: This can be interpreted as a type of a phishing attack. In solidity, `tx.origin` and `msg.sender` are separate. The account calling a contract is defined by `msg.sender`. `tx.origin` is the original sender of the transaction, which might lead to a string of other calls. However, if `tx.origin` is used for authorization, and the actual owner is conned to call a malicious contract which in turn calls the victim contract, then the authorization fails.

3. **Unprotected Ether Withdrawal:** Due to missing or inadequate access control mechanisms, it might be the case that anyone is able to withdraw Ether from the contract which is highly undesirable.
4. **Unprotected Selfdestruct:** `selfdestruct` kills a contract on the blockchain and send the contract balance to the specified address. Opcode `SELFDESTRUCT` is one of the few operations that costs negative gas as it frees up space on the blockchain. This construct is important because contracts may need to be killed if they are no longer required or if some bug is discovered. However, if this construct is put without proper protection mechanisms in place then anyone can kill the contract.
5. **Unexpected Ether:** Usually, when you send ether to a contract, its fallback function is executed. However, if the transfer of ether happens as a result of a `selfdestruct()` call, then the fallback function is not called. Therefore, a contract's balance should never be used in an if condition as it can be manipulated by a malicious user.
6. **Function and Variable Visibility:** Solidity has four visibility specifiers for functions and variables. However, being declared `public` is the most tricky from a security standpoint. If an important function like a payable function or a constructor with a wrong name is declared as public, then it can cause great monetary losses. This was observed in the Rubixi Contract which was the copy of the DynamicPyramid contract. However, the constructor name was not changed, making the original constructor a public function which anyone could call. This constructor decided the owner of the contract. Since anyone could now become the owner of the contract, it was compromised. Variable visibility does not have such drastic consequences as public variables get a public getter function. The Parity wallet attack also consisted of a function visibility bug.
7. **Integer Overflow and Underflow:** Solidity can handle up to 256 bit numbers, and therefore increasing (or decreasing) a number over (or below) the maximum (or minimum) value can result in overflows (or underflows). It is recommended to use OpenZeppelin's `SafeMath` library to mitigate such attacks.
8. **Variable Shadowing:** Variable shadowing occurs when a variable with the same name can be declared again. This can happen in case of a single contract (at the contract and function level) and also with multiple contracts. For example, a contract A inherits B, but both contracts have declared a variable x.
9. **Exception Handling:** Like in any object oriented programming language, exceptions may arise due to many reasons. These must be properly handled at the programmer level. Also, lower level calls do not throw an exception. They simple return a false value which needs to be checked and the exception should be handled manually.
10. **Denial of Service:** A denial of service attack from a smart contract's perspective happens when a smart contract becomes inaccessible to its users. Common reasons include failure of external calls or gas costly programming patterns.

11. **Call to the Unknown:** Ethereum Smart Contracts can make calls to other smart contracts. If the addresses of these smart contracts may be user provided then a malicious actor can utilize improper authentication to call a malicious contract. If the address is hard-coded, then it does not give the flexibility to update the contract to be called over time. Another issue is a special method called `delegatecall`. This makes the dynamically loaded code run in the caller's context. Therefore, if a `delegatecall` is made to a malicious contract, they can change storage values and potentially drain all funds from the contract.
12. **Bad Randomness:** Online games and lotteries are common dApp use cases. For these applications, a common choice for the seed of the random number generator is the hash or timestamp of some block that appears in the future. This is considered secure as the future is unpredictable. However, a malicious attacker can bias the seed in his favour.
13. **Untrustworthy Data Feeds:** In context of blockchains, a data feeds (also referred to as oracle) is an agent that verifies the integrity of the information before putting it on the blockchain. Once the data is published on the chain, its integrity can be guaranteed. However, the problem of making sure that the data feeds themselves are trustworthy is an active research problem.
14. **Transaction Order Dependence:** The order in which the transactions are picked up by miners might not be the same as the order in which they arrive. This creates a problem for contracts that rely on the state of the storage variables. Gas sent is usually important as it plays an important role in determining which transactions are picked first. A malicious transaction might be picked first, causing the original transaction to fail. This kind of race-condition vulnerability is referred to as transaction order dependence.
15. **Timestamp Dependence:** A lot of applications have a requirement to implement a notion of time in their applications. The most common method of implementing this is using the `block.timestamp` either directly or indirectly. However, a malicious miner with a significant computational power can manipulate the timestamp to get an output in his/her favour.

3 New Taxonomy for Ethereum Smart Contract Vulnerabilities

3.1 Existing Taxonomies and the Need for a New Taxonomy

The first taxonomy for smart contract vulnerabilities was given by Atzei et al. [5]. They divided the vulnerabilities into three broad categories based on their source. This included Solidity, EVM and blockchain. The vulnerabilities discussed did not give a holistic picture as it did not even contain common vulnerabilities like access control, function visibility, and transaction order dependence. These vulnerabilities have been proven to be quite disastrous as shown in the infamous Parity bug. Also, it was felt that only one level of hierarchy was less for proper analysis. Dika [21] in his Master's Thesis addressed many of the shortcomings of

the Atzei taxonomy. More vulnerability categories were added and an associated severity level was also given for each vulnerability. However, the single level hierarchy was carried forward from the previous work. Also, we noticed that some vulnerability classes do not pose an immediate security risk. For example, use of `tx.origin` was labelled as a vulnerability. However, just using `tx.origin` does not cause a security breach. The problem occurs when it is used for authorization. Similarly, `blockhash` may cause a security vulnerability if used as a source of randomness. However just using it in the code does not make a contract vulnerable. Because of these issues in the existing work, we felt that there was a need for an improved taxonomy that was more hierarchical - for better analysis and understanding. Also, issues of improper vulnerability naming and incomplete vulnerability listing also needed refinement.

3.2 A New Taxonomy of Ethereum Smart Contract Vulnerabilities

Based on our research and study of Ethereum smart contract vulnerabilities as discussed in Sect. 1, we have come up with a new and unified vulnerability taxonomy as shown in Table 1. With this new taxonomy, we try to overcome the problems in the existing literature. We try to cover almost all the security vulnerabilities that have been reported. Since, these are usually reported under different names, a security analyst would find that he/she is able to put any *existing* vulnerability he/she encounters under one of the many categories we have created. Also, unlike previous works, we have tried to eliminate any redundancies and/or incorrect categorizations. The taxonomy is hierarchical and therefore analysis using this taxonomy would give the security researcher better insights into the root security issues in smart contracts.

Based on the existing literature [21] and the OWASP Risk Rating Methodology [50], the severity levels are color-coded. The authors in this work categorized various severity levels according to their criticality level - high, medium, and low. In our work, the severity level is color-coded with red being high, orange being medium, and green being low.

4 Vulnerability Benchmark

4.1 Need for a Benchmark

It is observed that many security tools have come up for Ethereum smart contracts over the years. However, it is also observed that these tools are usually tested on different test-instances and in some cases even the ground truth is unknown. Therefore, as a smart contract developer or a user, it becomes difficult to actually compare the performance of different tools without a proper benchmark.

Dika [21] tried to solve this issue. However, he tested only three tools on just 23 vulnerable and 21 audited-safe contracts. A contract was called vulnerable if it had *any* vulnerability. However, it was not checked that a tool properly detected the vulnerability claimed and the results were presented as is.

Table 1. A new taxonomy of ethereum smart contract vulnerabilities

Solidity	Re-entrancy	
	Access Control	Protection Issues
		Authorization through `tx.origin`
		Unprotected Ether Withdrawal
		Unprotected `selfdestruct`
		Unexpected Ether
		Visibility Issues
		Function Visibility
		Variable Visibility
	Arithmetic Issues	Integer Overflow & Underflow
		Floating Point & Precision
	Solidity Programming Issues	Uninitialized Storage Pointers
		Variable Shadowing
		Keeping Secrets
		Type Casts
		Lack of Proper Signature Verification
		Write to Arbitrary Storage Location
		Incorrect Inheritance Order
		Typographical Errors
		Use of Assembly
		Use of Deprecated Functions/Constructions
		Floating or No Pragma
		Outdated Compiler Version
	Exception Handling	Unchecked Call
		Gasless Send
		Call Stack Limit
		Assert Violation
		Requirement Violation
	Call to the Unknown	Dangerous Delegate Call
		External Contract Referencing
	Denial of Service	DoS with block gas limit
		DoS with failed call
EVM	Short Address Attack	
	Immutable bugs	
	Stack size limit	
Blockchain	Bad Randomness	
	Untrustworthy Data Feeds	
	Transaction Order Dependence	
	Timestamp Dependence	
	Unpredictable state (Dynamic Libraries)	

4.2 Benchmark Creation Methodology

To create the benchmark, we collected contracts known to be vulnerable from various sources. This included:

– Smart Contract Weakness Classification (SWC) Registry [16]
– (Not So) Smart Contracts [17]
– EVM Analyzer Benchmark Suite [14]
– Research papers, theses and books [3–6,21]
– Various blog posts, articles, etc. [8,11,24,26,33,37,42,43,47,52]

After collecting all the instances, we manually removed the duplicate contracts - this was important as we found that there was notable overlap between contracts gathered from different sources. After this, we manually checked the contracts and classified them as per the new taxonomy. Finally we compiled the smart contracts into run-time bytecode. However, as each contract required a different version of solidity, and `solc-select` [20] did not support such a large range of compiler versions, we leveraged Remix IDE [25] to manually generate the run-time byte-codes for each contract and stored it separately. This was not done for the on-chain contracts and the run-time byte-codes for these were directly taken from the blockchain. A summary of the benchmark creation methodology is depicted in Fig. 1.

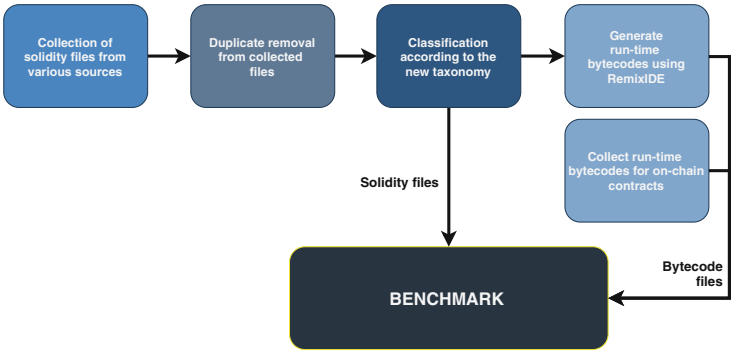

Fig. 1. Vulnerability benchmark creation

4.3 Benchmark Statistics

Unlike the previous work by Dika where only 23 vulnerable contracts were used, we have identified 162 unique vulnerable contracts. We have collected 34 on-chain vulnerable contracts. A few of them are - `SmartBillions, Lottery, EthStick, UGToken`, etc. including infamous contracts like the `DAO, Parity Wallet, Rubixi and King of the Ether Throne`.

It has been observed that Ethereum smart contracts have been used for creating ponzi-schemes to scam innocent people into loosing money by promising extraordinarily high returns [6]. Even though a ponzi contract might not be a direct security vulnerability, we have included them in our study because of the high monetary impact of such contracts. Apart from ponzi schemes like GovernMental, FirePonzi and Rubixi which have already been included, we added ponzi

schemes corresponding to the vulnerabilities - `Does not Refund, Allow Owner to withdraw funds` from contracts namely - `DynamicPyramid, GreedPit, NanoPyramid, Tomeka, ProtectTheCastle, and EthVentures` that exhibited one or more of the following properties - contracts that do not refund and contracts that allow the owner to withdraw funds. These properties are typical in ponzi schemes, however they cannot be classified as security vulnerabilities directly without knowing the context.

The final benchmark consists of 180 contracts spread over all the categories. Out of these we have 162 unique contracts. This includes 40 on-chain contracts (including six additional ponzi schemes). This is very high in comparison to the 23 vulnerable smart contracts identified by [21].

5 Study of Security Tools

5.1 Tools Available for Ethereum Smart Contracts

There are many different tools available for Ethereum Smart Contracts. These tools have been gathered from research publications and through Internet searches. In this section, we classify the various tools available into different categories, so that the end users can easily find which tool to use for their particular application. Even though our work is primarily concerned with Security Tools, the other tools are included for the reader's convenience.

1. **Security Tools:** These are tools which take as input either the source code or the bytecode of a contract and give outputs on the security issues present. These are the tools that we are primarily concerned in with our work. Examples of security tools include Mythril [15], and Securify [49].
2. **Visualization Tools:** Visualization tools help give graphical outputs like control flow graphs, dependency graphs, etc. of the given contract to help in analysis. Tools like solgraph [44] and rattle [18] fall under this category.
3. **Disassemblers and Decompilers:** A dis-assembler converts the binary code back into the high level language code while a decompiler converts the binary code to a low level language for better understanding. evm-dis [30] is a popular dis-assembler for smart contracts.
4. **Linters:** Linters are static analysis tools primarily focused on detecting poor coding practices, programming errors, etc. Ethlint [22] is a common linting tool of Ethereum smart contracts.
5. **Miscellaneous Tools:** This includes tools like SolMet [27] which help give common code metrics like number of lines of code, number of functions per contract, etc. for solidity source files.

5.2 Methods Employed by the Security Tools

1. **Static Analysis:** Static Analysis essentially means evaluating the program code without actually running it. It looks at the code structure, the decompiled outputs, and control flow graphs to identify common security issues.

SmartCheck [48], Slither [19] and Remix IDE [25] are static analysis security tools for Ethereum smart contracts.
2. **Symbolic Execution:** Symbolic execution is considered to be in the middle of static and dynamic analysis. It explores possible execution paths for a program without any concrete input values. Instead of values, it uses symbols and keeps track of the symbolic state. It leverages constraint solvers to make sure that all the properties are satisfied. Mythril [15] and Oyente [36] are the popular Symbolic Execution tools for smart contract security.
3. **Formal Verification:** Formal Verification incorporates mathematical models to make sure that the code is free of errors. Bhargavan et al. [7] conducted a study of smart contracts using F*. However, the work is not available as open source to the best of our knowledge.

5.3 Experimental Setup

For the purpose of the study, we select the security tools that are actively maintained, open-sourced, ready for use and cover a fairly large section of the vulnerabilities. Keeping the above constraints in mind, the following tools were selected -

1. **Remix IDE:** Remix IDE [25] is primarily an integrated development environment (IDE) for developing Solidity smart contracts. It can connect to the Ethereum network using Metamask and developers can directly deploy smart contracts from Remix. It is developed and maintained by the Ethereum Foundation. The IDE has a security module to help developers with common security issues like re-entrancy, etc. It requires the solidity file of the contract to work. As a web interface was available, the testing using the benchmark instances was carried out manually.
2. **SmartCheck:** SmartCheck [48] is a static analysis tool for Solidity and Vyper smart contracts. It is developed by SmartDec and the University of Luxembourg. Like other static analysis tools, it does not work on byte-codes and requires the source codes to be present for analysis. It works by transforming the source codes into an intermediate representation which is XML-based. This representation is then checked against XPath patterns to highlight potential vulnerabilities in the code. The tool is open sourced and also has a web interface hosted at [45].
3. **Slither:** Slither [19] is a static analysis tool for solidity source files written in Python 3. It is open sourced and is developed by Trail of Bits. It works on contracts written in solidity >= 0.4 and requires the solidity files for analysis. It leverages an intermediate representation call SlithIR for code analysis. However, it requires the correct solidity version to be installed in the system. For this, we utilize another tool by Trail of Bits called `solc-select` [20] to switch to the right compiler version which is predetermined manually.
4. **Oyente:** Oyente [36] is one of the earliest security tools for Solidity smart contracts. It was developed by security researchers at the National University of Singapore and is now being maintained by Melonport. Oyente leverages

symbolic execution to find potential vulnerabilites in the smart contracts. It works with both byte-codes and solidity files. Being one of the first tools in this area, Oyente has been extended by many researchers over the years. For example, the control flow graphs generated by Oyente are also used by EthIR [1], which is a high level analysis tool for Solidity. A web interface for the tool is also available [38].

5. **Securify:** Securify [49] has been created by researchers at ETH Zurich in collaboration with ChainSecurity for security testing of Ethereum smart contracts. It works on both solidity source files and byte-codes. It has also received funding from the Ethereum Foundation to help mitigate the security issues in smart contracts. It analyzes the contract symbolically to get semantic information and then checks against patterns to see if a particular property holds or not. A web interface is also available at [10].
6. **Mythril:** Mythril [15] is a security tool developed by ConsenSys. It uses as a combination of symbolic execution and taint analysis to identify common security issues. Recently, a new initiative called MythX was launched with a similar core as Mythril for smart contract developers to provide security as a service. However, it is still in beta testing and is not available as open source. Therefore, we use Mythril Classic for our testing purposes.

Table 2 summarizes the tools selected for the study along with versions of the tools used. Table 3 shows the vulnerability coverage as claimed by the tools. According to the claims, we observe that the vulnerability coverage across all the tools is fairly good with most vulnerabilities being covered by one tool or another. All the experiments were carried out on a machine running Ubuntu 18.04.2 LTS on an Intel® Core™ i7-4770 CPU with 16 GB DDR3 RAM. Also, the tools that worked on both solidity and bytecode files were tested on bytecode files only. The results output by each tool were then converted to the new taxonomy as shown in Table 3 to allow us to compare the tools uniformly.

Table 2. Summary of tools used in the study

	Remix IDE	Smart-Check	Slither	Oyente	Securify	Mythril
Version/ Date Used	4-Mar-2019	2.0.1	0.4.0	0.2.7	17-Apr-19	0.20.4
Technique	Static Analysis	Static Analysis	Static Analysis	Symbolic Execution	Symbolic Execution	Symbolic Execution
WUI/ CLI	WUI	WUI + CLI	CLI	WUI + CLI	WUI + CLI	WUI + CLI
Works on src-file/ bytecode	src-file	src-file	src-file	src-file + bytecode	src-file + bytecode	src-file + bytecode
Developed by	Ethereum Foundation	SmartDec	Trail of Bits	NUS + Melonport	ETH Zurich	ConsenSys

Table 3. Vulnerability mapping to the new taxonomy

Reported by the Tool	Mapping to the new Taxonomy
Remix IDE	
Transaction origin	Authorization through `tx.origin`
Check-effects	Re-entrancy
Block timestamp usage	Timestamp Dependence
`block.blockhash` usage	Bad randomness
inline assembly	Use of Assembly
Use of `selfdestruct`	Unprotected `selfdestruct`
Low level calls/use of send	Unchecked Call
SmartCheck	
Deprecated Constructions	Use of Deprecated Functions/Constructions
Gas limit in loops	DoS with block gas limit
Upgrade to 0.5.0	Outdated Compiler Version
Pragmas version	Floating or No Pragma
Send, Unchecked call, Call without data	Unchecked Call
Using inline assembly	Use of Assembly
Incorrect Blockhash	Bad Randomness
Transfer in loop	DoS with failed call
Exact time	Timestamp dependence
Div mul	Floating Point and Precision
Visibility	Function Default Visibility
Locked money	Ponzi Scheme – Do Not Refund
Redundant fallback reject, Balance equality	Unexpected Ether
Array length manipulation	Write to Arbitrary Storage Location
Slither	
Reentrancy-eth,reentrancy-no-eth,reentrancy-benign	Re-entrancy
tx-origin	Authorization through tx.origin
timestamp	Timestamp dependence
Uninitialized-state, uninitialized-local, uninitialized-storage	Uninitialized storage pointers
suicidal	Unprotected selfdestruct
assembly	Use of Assembly
deprecated-standards	Use of Deprecated Functions or Constructions
solc-version	Outdated Compiler Version
calls-loop	Denial of Service with failed call
arbitrary-send	Unprotected Ether Withdrawal
incorrect-equality	Unexpected Ether
Unused-return, low-level-calls	Unchecked External Call
Shadowing-builtin, shadowing-local, shadowing-state	Shadowing State Variables
controlled-delegatecall	Dangerous Delegate Call
locked-ether	Ponzi scheme – Does not Return
OYENTE	
Call stack	Stack size limit
Re-entrancy	Re-entrancy
Time Dependency	Timestamp Dependence
Integer Overflow, Integer Underflow	Integer Overflow & Underflow
Money Concurrency	Transaction Order Dependence
Mythril Classic	
Integer Underflow, Integer Overflow	Integer Overflow & Underflow
Unchecked Call Return Value	Unchecked Call
Unprotected Selfdestruct	Unprotected selfdestruct
Unprotected Ether Withdrawal	Unprotected Ether Withdrawal
Use of tx.origin	Authorization through tx.origin
Exception State	Exception Handling
External Call To Fixed/User-Supplied Address	Dangerous Delegate Call
Use of callcode	Use of Deprecated Functions/Constructs
Dependence on predictable variable/environment variable	Bad Randomness
Multiple Calls in a Single Transaction	Denial of Service
Securify	
DAO, DAOConstantGas	Re-entrancy
LockedEther	Ponzi Scheme – Do not Refund
MissingInputValidation	Type Casts
RepeatedCall	Dangerous Delegate Call
TODAmount, TODReceiver	Transaction Ordering Dependence
UnhandledException	Unchecked Call
UnrestrictedEtherFlow	Unprotected Ether Withdrawal
UnrestrictedWrite	Write to arbitrary storage location

6 Results

For each tool, we run it against the benchmark. Then, we identify the relevant entries using the Table 4 to identify the vulnerabilities which the tool claims to identify. We then, map the results using the mapping in Table 3 and present the results in a tabular format in Table 5 and Table 6. The table for each tool depicts the vulnerable contracts it detected successfully and correctly, the contracts it could not detect correctly, and the contracts on which the tool could not finish it's evaluation because of some error or exception being raised. Securify is the only tool in our study that marks a contract as 'safe' from a vulnerability. If a vulnerable contract was wrongly labelled as 'safe', we call it a false negative.

Table 4. Tool-vulnerability matrix as claimed by the tools

		Remix	Slither	SmartCheck	Oyente	Mythril	Securify	SUM
SOLIDITY	Re-entrancy	Y	Y		Y		Y	4
	Authorization through `tx.origin`	Y	Y			Y		3
	Unprotected Ether Withdrawal		Y			Y	Y	3
	Unprotected `selfdestruct`	Y	Y			Y		3
	Unexpected Ether		Y	Y				2
	Function Visibility			Y				1
	Variable Visibility							0
	Integer Overflow & Underflow				Y	Y		2
	Floating Point & Precision			Y				1
	Uninitialized Storage Pointers		Y					1
	Variable Shadowing		Y					1
	Keeping Secrets			Y				1
	Type Casts						Y	1
	Lack of Proper Signature Verification							0
	Write to Arbitrary Storage Location			Y			Y	2
	Incorrect Inheritance Order							0
	Typographical Errors							0
	Use of Assembly	Y	Y	Y				3
	Use of Deprecated Functions		Y	Y		Y		3
	Floating or No Pragma			Y				1
	Outdated Compiler Version		Y	Y				2
	Unchecked Call	Y	Y	Y			Y	4
	Gasless Send						Y	1
	Call Stack Limit						Y	1
	Assert Violation						Y	1
	Requirement Violation						Y	1
	Dangerous Delegate Call		Y			Y	Y	3
	External Contract Referencing							0
	DoS with block gas limit			Y			Y	2
	DoS with failed call		Y	Y			Y	3
EVM	Short Address Attack							0
	Immutable bugs							0
	Stack size limit				Y			1
B/CHAIN	Bad Randomness	Y		Y		Y		3
	Untrustworthy Data Feeds							0
	Transaction Order Dependence				Y		Y	2
	Timestamp Dependence	Y	Y	Y	Y			4
	Unpredictable state							0
PS	Does not Return		Y	Y			Y	3
	Allows Owner to Withdraw Funds							0
	TOTAL	7	15	15	5	7	14	

1. **Remix IDE:** The performance of Remix IDE is surprisingly good. As seen in Table 6, it detects vulnerabilities like `tx.origin` authorization, use of assembly, unchecked call and timestamp dependence with 100% accuracy. However,

we find that these vulnerabilities are caught by mere presence of certain constructs without checking whether they actually result in a vulnerability or not. For example, Timestamp Dependence flag is raised if `timestamp` is used anywhere in the code. Similarly, `tx.origin` flag is raised if `tx.origin` is used anywhere within the code without checking if any it causes any security issue or not. The `selfdestruct` module works similarly. However it could not detect the Parity Bug because it uses the older `suicide` construct. It was also observed that for solidity versions 0.3.1 and prior, the check-effects and the selfdestruct modules gave an error. This resulted in the famous DAO contract not being analysed by the tool.

2. **SmartCheck:** The performance of SmartCheck is given in Table 6. It has a good performance in only a few of the many categories that it can detect. They include security issues like use of deprecated functions, unchecked call, use of assembly, etc. However, the performance on other instances is not very good.
3. **Slither:** Slither has a very good performance across most of the categories as shown in Table 6. There was no category that it could not detect even one instance from. The biggest drawback of slither is that does not work with older solidity versions (prior to 0.4) and requires the correct version of solidity to be present on the system. Because of this, a lot of contracts in the benchmark gave errors with slither. However, it is a very good tool for smart contract developers who are developing in newer versions of solidity.
4. **Oyente:** Being one of the earliest tools, Oyente is now showing it's age. It covers a very low number of vulnerabilities. The results are shown in Table 6. Average EVM code coverage for the entire benchmark set was found to be 75.98%. Also, there was not a single report of integer overflow or underflow across the complete benchmark. We believe this is some bug in the tool causing this behaviour.
5. **Securify:** Securify is the only tool that reports a contract as 'safe' from a particular vulnerability. If the contract contains a vulnerability, and Securify reports it as 'safe', we call it false negative. From Table 5 we can see that Securify reports a lot of false negatives. However, it has a decent performance on re-entrancy bug detection.

Table 5. Results of securify on the vulnerability benchmark

Vulnerability	Total	Detected	Not Detected	Error	False Negative
Re-entrancy	10	6	0	2	2
Transaction Ordering Dependence	9	2	2	0	5
Dangerous Delegate Call	6	0	0	0	6
Unchecked Call	3	0	0	0	3
Type Casts	6	1	4	0	1
Ponzi – Do not Refund	4	0	0	0	4
Unprotected Ether Withdrawal	7	0	3	0	4
Write to arbitrary storage	2	0	2	0	0

Table 6. Results of various tools on the Vulnerability Benchmark

Results of Remix IDE on the Vulnerability Benchmark				
Vulnerability	**Total**	**Detected**	**Not Detected**	**Error**
Auth through tx.origin	2	2	0	0
Use of Assembly	1	1	0	0
Timestamp Dependence	6	6	0	0
Unchecked Call	3	3	0	0
Unprotected selfdestruct	3	2	1	0
Re-entrancy	10	5	3	2
Bad Randomness	11	4	7	0
Results of Oyente on the Vulnerability Benchmark				
Vulnerability	**Total**	**Detected**	**Not Detected**	**Error**
Stack Size Limit	1	1	0	0
Re-entrancy	10	5	5	0
Timestamp Dependence	6	2	4	0
Transaction Order Dependence	9	5	4	0
Results of Slither on the Vulnerability Benchmark				
Vulnerability	**Total**	**Detected**	**Not Detected**	**Error**
Auth through tx.origin	2	2	0	0
Unprotected selfdestruct	3	2	0	1
Use of Assembly	1	1	0	0
Use of Deprecated Functions	1	1	0	0
Outdated Compiler Version	1	1	0	0
Unexpected Ether	2	2	0	0
Unchecked Call	3	1	0	2
Variable Shadowing	3	3	0	0
Uninitialized storage pointers	5	4	1	0
Dangerous Delegate Call	6	2	1	3
Re-entrancy	10	4	3	3
DoS with failed call	3	1	1	1
Timestamp dependence	6	1	2	3
Unprotected Ether Withdrawal	7	2	4	1
Ponzi – Does not Return	4	0	0	4

Results of SmartCheck on the Vulnerability Benchmark				
Vulnerability	**Total**	**Detected**	**Not Detected**	**Error**
Outdated Compiler Version	1	0	0	1
Use of Deprecated	1	1	0	0
Unchecked Call	3	3	0	0
Use of Assembly	1	1	0	0
Function Visibility	12	9	3	0
Unexpected Ether	2	1	1	0
DoS with block gas limit	5	1	4	0
Time stamp dependence	6	1	5	0
Floating or No Pragma	2	0	1	1
Bad Randomness	11	0	10	1
DoS with failed call	3	0	3	0
Floating Point and Precision	2	0	2	0
Ponzi – Do Not Refund	4	0	4	0
Write to Arbitrary Storage	2	0	2	0
Results of Mythril on the Vulnerability Benchmark				
Vulnerability	**Total**	**Detected**	**Not Detected**	**Error**
Unchecked Call	3	1	0	2
Auth through tx.origin	2	1	0	1
Use of Deprecated Functions	1	1	0	0
Unprotected selfdestruct	3	2	1	0
Unprotected Ether Withdrawal	7	4	3	0
Integer Overflow & Underflow	31	11	10	10
Dangerous Delegate Call	6	2	2	2
Exception Handling	29	12	14	3
Bad Randomness	11	1	4	6
Denial of Service	4	0	4	0
-	-	-	-	-
-	-	-	-	-
-	-	-	-	-

6. **Mythril:** The performance of Mythril on the benchmark is shown in Table 6. It is able to detect the attacks with a fair accuracy, however it encounters a lot of errors. This makes its performance inferior to some static analysis tools like Slither.

The effectiveness of the tools in detecting the vulnerabilities in the benchmark is shown in Table 7. The cells highlighted in green indicate that all the instances of that vulnerability present in the benchmark are successfully detected, while grey highlights the maximum vulnerabilities (though not all) accurately detected across all the tools. We observe that many vulnerabilities are not being detected by the tools. We also observe that even though the tools cover a wide spectrum of vulnerabilities, they are not very accurate in detecting them. The best tool from our study is Slither. It covers a wide range of vulnerabilities and is the only tool that detected at-least one from each category it could successfully evaluate. The only drawback is that it works on solidity versions greater than 0.4.0. Nevertheless, it is still a good tool for new smart contract developers.

7 Related Work

Atzei et al. [5] conducted the first survey of attacks on Ethereum smart contracts and also gave the first taxonomy of Ethereum smart contract vulnerabilities. They also look at some of the popular vulnerable contracts like the DAO, Rubixi, GovernMental and King of the Ether throne.

Table 7. Tool effectiveness for different vulnerabilities

		RemixIDE	Slither	SmartCheck	Oyente	Mythril	Securify
SOLIDITY	Re-entrancy	5	4		5		6
	Authorization through `tx.origin`	2	2			1	
	Unprotected Ether Withdrawal		2			4	0
	Unprotected `selfdestruct`	2	2			2	
	Unexpected Ether		2	1			
	Function Visibility			9			
	Variable Visibility						
	Integer Overflow & Underflow				-	11	
	Floating Point & Precision			0			
	Uninitialized Storage Pointers		4				
	Variable Shadowing		3				
	Keeping Secrets						
	Type Casts						1
	Lack of Proper Signature Verification						
	Write to Arbitrary Storage Location			0			0
	Incorrect Inheritance Order						
	Typographical Errors						
	Use of Assembly	1	1	1			
	Use of Deprecated Functions/Constructions		1	1		1	
	Floating or No Pragma			0			
	Outdated Compiler Version		1	0			
	Unchecked Call	3	1	3		1	0
	Gasless Send					0	
	Call Stack Limit					0	
	Assert Violation					12	
	Requirement Violation					0	
	Dangerous Delegate Call		2			2	0
	External Contract Referencing						
	DoS with block gas limit			1			0
	DoS with failed call		1	0			
EVM	Immutable bugs						
	Stack size limit				1		
B/CHAIN	Bad Randomness	4		0		1	
	Transaction Order Dependence				5		2
	Timestamp Dependence	6	1	1	2		
	Unpredictable state (Dynamic Libraries)						
PS	Does not Return		0	0			0
	Allows Owner to Withdraw Funds						

Dika [21] in his master's thesis, extended the taxonomy given by Atzei et al. [5]. He also tested the effectiveness of three security tools on a data-set of 23 vulnerable and 21 safe contracts. It is observed that the data-set and the number of tools used for the study is quite less. Also, the taxonomy needs hierarchy for better analysis. Mense et al. [39] look at the security analysis tools available for Ethereum smart contracts and cross reference them to the extended taxonomy given by Dika [21] to identify the vulnerabilities captured by each tool. However, the tool's effectiveness in catching those vulnerabilities is not studied. Buterin [8] in his post outlines the various vulnerable smart contracts with an elementary categorization. He also emphasises the need to experiment with various tools and standardization wherever possible to mitigate bugs in smart contracts. Angelo et al. [2] surveyed the various tools available to Ethereum smart contract developers. They do a very broad categorization of tools - those which are publicly available and those which are not publicly available. Antonopoulos et al. [3] in their book on Ethereum have dedicated a chapter on smart contract security. They cover the various vulnerabilities encountered by smart contract developers and give real world examples and preventative techniques. It is a good reference for smart contract developers.

8 Conclusion

Security researchers and smart contract developers face three problems when dealing with smart contracts - lack of an updated and organized study of the possible vulnerabilities and their causes, lack of a standard taxonomy and naming convention of these vulnerabilities and lack of a benchmark to compare and evaluate the performance of the different tools available for smart contract security, so that they can make an informed decision about which tool to use. In this work, we conduct an organized study of smart contract vulnerabilities and develop a novel taxonomy that is hierarchical and uses nomenclature used popularly by security researchers. We also develop a comprehensive vulnerability benchmark containing 180 vulnerable contracts across different vulnerability categories. This benchmark is based on the novel taxonomy explained in this work. Finally, we compare and analyze the performance of different security tools using the benchmark. We observe that the static analysis tools perform better than the symbolic execution tools. As this is an active research area, updation of the benchmark and the taxonomy is needed from time to time. Also, to detect false positives, we may develop a non-vulnerable benchmark that contains instances that might seem vulnerable at the first glance but do not pose a security risk. It would be interesting to see the performance of the tools on such instances.

References

1. Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I.: EthIR: a framework for high-level analysis of ethereum bytecode. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 513–520. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_30
2. Di Angelo, M., Antipolis, S.: A survey of tools for analyzing ethereum smart contracts (2019)
3. Antonopoulos, A.M., Wood, G.: Mastering Ethereum: Building Smart Contracts and Dapps. O'Reilly Media, Sebastopol (2018)
4. Atzei, N., Bartoletti, M., Cimoli, T.: Attacks - A Survey of Attacks on Ethereum Smart Contracts. http://blockchain.unica.it/projects/ethereum-survey/attacks.html. Accessed 2 May 2019
5. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
6. Bartoletti, M., Carta, S., Cimoli, T., Saia, R.: Dissecting ponzi schemes on ethereum: identification, analysis, and impact. arXiv preprint arXiv:1703.03779 (2017)
7. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, pp. 91–96. ACM (2016)
8. Buterin, V.: Thinking About Smart Contract Security. https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/. Accessed 2 May 2019
9. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. White paper (2014)

10. ChainSecurity. Securify Scanner for Ethereum Smart Contracts. https://securify.chainsecurity.com. Accessed 16 May 2019
11. CityMayor. How Someone Tried to Exploit a Flaw in Our Smart Contract and Steal All of Its Ether. https://blog.citymayor.co/posts/how-someone-tried-to-exploit-a-flaw-in-our-smart-contract-and-steal-all-of-its-ether/. Accessed 2 May 2019
12. Ethereum Community. Ethereum Homestead Documentation. http://ethdocs.org/en/latest/index.html. Accessed 10 May 2019
13. ConsenSys. Ethereum Smart Contract Best Practices - Known Attacks. https://consensys.github.io/smart-contract-best-practices/known_attacks/. Accessed 24 April 2019
14. Consensys. EVM Analyzer Benchmark Suite. https://github.com/ConsenSys/evm-analyzer-benchmark-suite. Accessed 2 May 2019
15. ConsenSys. Mythril Classic. https://github.com/ConsenSys/mythril-classic. Accessed 16 May 2019
16. Consensys. Smart Contract Weakness Classification and Test Cases. https://smartcontractsecurity.github.io/SWC-registry/. Accessed 2 May 2019
17. Crytic. (Not So) Smart Contracts. https://github.com/crytic/not-so-smart-contracts Accessed 2 May 2019
18. Crytic. rattle. https://github.com/crytic/rattle. Accessed 16 May 2019
19. Crytic. Slither, the Solidity source analyzer. https://github.com/crytic/slither. Accessed 2 May 2019
20. Crytic. solc-select. https://github.com/crytic/solc-select. Accessed 2 May 2019
21. Dika, A.: Ethereum smart contracts: Security vulnerabilities and security tools. Master's thesis, NTNU (2017)
22. Dua, R.: EthLint. https://github.com/duaraghav8/Ethlint. Accessed 16 May 2019
23. Eyal, I., Sirer, E.G.: Majority is not enough: bitcoin mining is vulnerable. Commun. ACM **61**(7), 95–102 (2018)
24. Falkon, S.: The Story of the DAO - Its History and Consequences. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee. Accessed 2 May 2019
25. Ethereum Foundation. Remix - Solidity IDE. https://remix.ethereum.org/. Accessed 2 May 2019
26. NCC Group. DASP - TOP 10. https://dasp.co/index.html. Accessed 2 May 2019
27. Hegedus, P.: Towards analyzing the complexity landscape of solidity based ethereum smart contracts. Technologies **7**(1), 6 (2019)
28. Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on bitcoin's peer-to-peer network. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 129–144 (2015)
29. Hollander, L.: The Ethereum Virtual Machine - How does it work? https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e. Accessed 10 May 2019
30. Johnson, N.: evmdis. https://github.com/arachnid/evmdis. Accessed 16 May 2019
31. Karame, G.O., Androulaki, E.: Bitcoin and Blockchain Security. Artech House (2016)
32. Karame, G.O., Androulaki, E., Capkun, S.: Double-spending fast payments in bitcoin. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 906–917. ACM (2012)
33. KingoftheEther. Post-Mortem Investigation, February 2016. https://www.kingoftheether.com/postmortem.html. Accessed 2 May 2019
34. Li, X., Jiang, P., Chen, T., Luo, X., Wen, Q.: A survey on the security of blockchain systems. Future Gen. Comput. Syst. **107**, 841–853 (2020). ISSN: 0167-739X

35. Lin, I.-C., Liao, T.-C.: A survey of blockchain security issues and challenges. IJ Netw. Secur. **19**(5), 653–659 (2017)
36. Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269. ACM (2016)
37. Manning, A.: Solidity Security: Comprehensive list of known attack vectors and common anti-patterns. https://blog.sigmaprime.io/solidity-security.html. Accessed 2 May 2019
38. Melonport. Oyente. https://oyente.melonport.com. Accessed 2 May 2019
39. Mense, A., Flatscher, M.: Security vulnerabilities in ethereum smart contracts. In: Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services, iiWAS 2018, pp. 375–380. ACM, New York (2018)
40. nick256. Smart Contract Security: Part 1 Reentrancy Attacks. https://hackernoon.com/smart-contract-security-part-1-reentrancy-attacks-ddb3b2429302. Accessed 24 Apr 2019
41. Niu, J., Feng, C.: Selfish Mining in Ethereum. arXiv e-prints, January 2019
42. PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts. https://blog.peckshield.com/2018/04/25/proxyOverflow/. Accessed 2 May 2019
43. Reutov, A.: Predicting Random Numbers in Ethereum Smart Contracts. https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620. Accessed 2 May 2019
44. Raine Revere. solgraph. https://github.com/raineorshine/solgraph. Accessed 16 May 2019
45. SmartDec. SmartCheck. https://tool.smartdec.net/. Accessed 2 May 2019
46. Szabo, N.: The idea of smart contracts. Nick Szabo's Papers and Concise Tutorials, 6 (1997)
47. Parity Technologies. Parity: Security Alert. https://www.parity.io/security-alert-2/. Accessed 2 May 2019
48. Tikhomirov, S., et al.: Smartcheck: static analysis of ethereum smart contracts. In: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 9–16. IEEE (2018)
49. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82. ACM (2018)
50. UcedaVelez, T.: OWASP Risk Rating Methodology. https://www.owasp.org/index.php?title=OWASP_Risk_Rating_Methodology&oldid=247702. Accessed 25 Apr 2019
51. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper (2014)
52. Yuan, M.: Building a safer crypto token. https://medium.com/cybermiles/building-a-safer-crypto-token-27c96a7e78fd. Accessed 2 May 2019