


## Review

# The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support

Haozhe Zhou, Amin Milani Fard \*  and Adetokunbo Makanju \*Department of Computer Science, New York Institute of Technology, Vancouver, BC V5M 4X3, Canada;  
hzhou25@nyit.edu

\* Correspondence: amilanif@nyit.edu (A.M.F.); amakanju@nyit.edu (A.M.)

**Abstract:** Smart contracts are self-executing programs that run on the blockchain and make it possible for peers to enforce agreements without a third-party guarantee. The smart contract on Ethereum is the fundamental element of decentralized finance with billions of US dollars in value. Smart contracts cannot be changed after deployment and hence the code needs to be verified for potential vulnerabilities. However, smart contracts are far from being secure and attacks exploiting vulnerabilities that have led to losses valued in the millions. In this work, we explore the current state of smart contracts security, prevalent vulnerabilities, and security-analysis tool support, through reviewing the latest advancement and research published in the past five years. We study 13 vulnerabilities in Ethereum smart contracts and their countermeasures, and investigate nine security-analysis tools. Our findings indicate that a uniform set of smart contract vulnerability definitions does not exist in research work and bugs pertaining to the same mechanisms sometimes appear with different names. This inconsistency makes it difficult to identify, categorize, and analyze vulnerabilities. We explain some safeguarding approaches and best practices. However, as technology improves new vulnerabilities may emerge. Regarding tool support, SmartCheck, DefectChecker, contractWard, and sFuzz tools are better choices in terms of more coverage of vulnerabilities; however, tools such as NPChecker, MadMax, Osiris, and Sereum target some specific categories of vulnerabilities if required. While contractWard is relatively fast and more accurate, it can only detect pre-defined vulnerabilities. The NPChecker is slower, however, can find new vulnerability patterns.

**Keywords:** smart contract; blockchain; security; solidity

**Citation:** Zhou, H.; Milani Fard, A.; Makanju, A. The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support. *J. Cybersecur. Priv.* **2022**, *2*, 358–378. <https://doi.org/10.3390/jcp2020019>

Academic Editor: Nour Moustafa

Received: 20 January 2022

Accepted: 23 May 2022

Published: 27 May 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Cryptocurrencies and decentralized finance (DeFi) feature the utilization of blockchain to transfer funds among peers on networks without intermediaries. DeFi proposes a model in which participants make deals through automated execution of blockchain-based smart contracts [1], and revolves around decentralized applications (DApps) that perform financial functions on immutable and publicly visible ledgers called blockchains. Smart contracts are programs that execute instructions in an automated manner once triggered. Most DApps are smart contracts that run on a blockchain network and leverage its consensus protocols to store their source codes and enforce clauses under predefined rules to render trustworthy services [2].

Ethereum [3] is the first and the most prominent platform that supports the implementation of smart contracts in a high-level programming language, such as Solidity [4]. Ethereum is the second-largest blockchain platform with a market capitalization of 317.3 Billion USD as of June 2021 [5] and provides a run-time environment for more than 95% of DeFi applications [6,7]. Bitcoin [8] supports the development and execution of smart contracts too, but the script language it uses has limitations, such as barely supporting transactions beyond verifying signatures [2]. While Bitcoin is just a cryptocurrency and payment system that facilitates payment processes, Ethereum is a computation platform [9]

supported by the Ethereum Virtual Machine (EVM), which is a state machine that creates a sandbox for executing smart contracts in which the state refers to the ledger.

The smart contract is a fundamental element of DeFi and hence its security is a matter of great importance. Smart contracts cannot be changed after deployment and thus the code need to be verified for potential vulnerabilities. Successful attacks on smart contracts have caused significant financial loss such as the well-known Decentralized Autonomous Organization (DAO) attack that evaporated millions of dollars and forced the Ethereum hard fork [10]. The immutability, decentralization, and publicly visible features of smart contracts not only lead to their wide adoption but also created pitfalls that prevent further adoption. Furthermore, the high-level programming language for writing smart contracts is in an early stage itself. Misunderstanding of best practices and mishandling of programming reflects the lack of knowledge about security issues among smart-contract developers.

In this paper, we investigate vulnerabilities that pertain to smart contracts by reviewing the latest work published since 2018 to complement previous studies and address the following research questions:

- **RQ1:** What vulnerabilities in Ethereum smart contracts have been recently studied?
- **RQ2:** What are possible countermeasures to mitigate smart-contract vulnerabilities?
- **RQ3:** What are the accuracy, efficiency, and limitations of existing security-analysis tools for smart contracts?

### 1.1. Methodology

We studies articles from conference proceedings, journals, preprints, theses, and online content to address our research questions. Since Ethereum is the most common platform for smart contract implementation, this study focuses on Ethereum vulnerabilities, countermeasures, and tool support. We searched Google Scholar, ACM Digital Library, IEEE Xplore Digital Library, Springer Online Library, and arXiv, to find literature related to Ethereum smart contract security. These databases are the most prevalent repositories for papers related to smart contracts. About 100 research articles were initially retrieved and studied, out of which around 50 papers were selected for the final survey based on our exclusion criteria. We excluded papers written in languages other than English and disregarded those unrelated to smart contracts or Ethereum. We then checked the relevance of collected documents to our research questions through their titles and abstracts. Next, we referred to the list of references of our selected papers and performed the same filtering process. We also acknowledge that some recent vulnerabilities, attacks, or tools may not have been reported in academic publications and hence we expanded our literature by searching online reports and articles through a Google search.

### 1.2. Contributions

In this work, we review the latest advancement and research work published since 2018 to complement studies such as [11–13]. Our survey also covers some missing analysis of more recent surveys on the Ethereum smart-contract security vulnerabilities, attacks, prevention, and tools [14–19]. We studied 13 vulnerabilities in Ethereum smart contract and their countermeasures, and investigate nine security-analysis tools. Our findings indicate that a uniform set of smart-contract vulnerability definitions does not exist in research work and bugs pertaining to the same mechanisms sometimes appear with different names. This inconsistency makes it difficult to identify, categorize, and analyze vulnerabilities. We explain some safeguarding approaches and best practices. However, as technology improves new vulnerabilities may emerge. With regards to tool support, SmartCheck, DefectChecker, contractWard, and sFuzz tools are better choices in terms of more coverage of vulnerabilities; however, tools such as NPChecker, MadMax, Osiris, and Sereum target some specific categories of vulnerabilities if required. While contractWard has good accuracy and fast process time, it can only detect specifically pre-defined vulnerabilities. The NPChecker has a slower processing time, but it may find new patterns for each vulnerability.

### 1.3. Organization

The rest of this paper is as follows. Section 2 introduces the technical background of the smart contract for later discussion. Section 3 reviews the literature on this topic and summarizes what has been carried out. Section 4 describes the smart contract vulnerabilities and their causes. Section 5 provides the results of our comparisons on security-analysis tools and updates on the status of security tools. Section 6 discusses the limitations and research questions. Finally, Section 7 draws conclusions and suggests future works.

## 2. Background

To better understand Ethereum and smart contracts, we explain Ethereum accounts, a smart contract's life cycle, and look deeper into its working mechanism.

### 2.1. Ethereum Accounts

The basic element of Ethereum is *accounts*, also called *account state*, and each account has four fields: *nonce*, *balance*, *storage*, and *code*. The nonce is a transaction counter, which means for every new transaction sent by this account, the nonce will be increased by one and attached in the transaction data structure; the balance is the amount of Ether (the currency used in the Ethereum platform) the account owns; the storage is memory space for code and its execution; the code is where the smart-contract code is stored [20].

There are two kinds of accounts: *external accounts* and *contract accounts*. The major difference between the two is whether the *code* field is empty or not. The *external accounts* are controlled by public-private key pairs (owned by human account holders) while the *contract accounts* are controlled by their code [20,21]. Both accounts are hashed and stored in a data structure called a modified Merkle Patricia tree [21], which has its root hashed and stored in every block. The *external accounts* could initiate an action that alters the state of the EVM, which is called a *transaction*. Transactions are broadcast to the whole network. A miner will later pick and execute transactions and propagate the resulting state change to the rest of the network. The execution of transactions is not free because the execution of transactions and the accompanying state change that must be universally accepted requires computing resources that consume a lot of energy. The *gas* is created for transaction execution and smart-contract interactions. Gas is units that the initiator needs to pay for the transaction executions; the gas limit is the maximum amount of gas that the initiator is willing to pay. The initiator also sets the *gas price*, which shows the amount of Ether that the initiator is willing to pay for each unit of gas. Usually, the higher the price a user is willing to pay, the higher chance the user's transactions are chosen to be executed by miners. A smart contract is a program that resides in a *contract account* and controls the behaviour of the contract account.

### 2.2. Life Cycle of Smart Contracts

The authors in [22] outline four different stages of the smart contract: creation, deployment, execution, and completion.

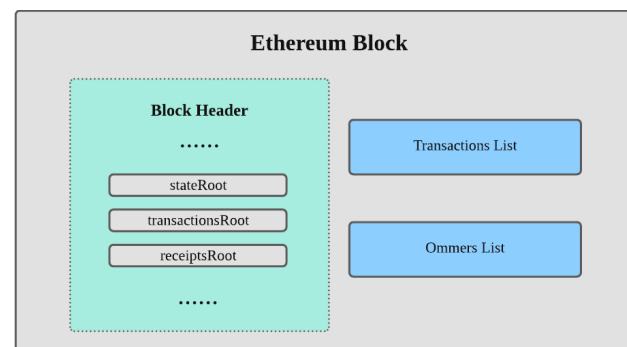
- **Creation.** In its creation stage, an EVM can be targeted by multiple high-level languages, such as Solidity [4], Serpent [23] or Bamboo [24]. The most commonly used one is Solidity. It is a JavaScript-like language and Turing-complete. EVM cannot run the Solidity codes directly, so they are compiled to opcodes (low-level instructions used by EVM) and encoded to bytecodes for storage reasons.
- **Deployment.** The Ethereum protocol outlines two kinds of transactions: one that invokes message calls, and the other that results in contract deployment [21]. The developer initiates a transaction that contains bytecodes (stored in a field called *init* in the transaction structure), and this action returns another fragment of the code that will be stored in the EVM running environment and will be executed later.
- **Execution.** While in the execution stage, a smart contract is a running program, like a process or thread in a stand-alone computer. It will receive transactions (the first type) and the data that will be passed to the contract as parameters. Then, the EVM

executes the instructions one by one until finished or the gas limitation is reached. This process happens at the time a new block is mined.

- **Completion.** After the execution, states are updated and stored in blockchains together with transactions. This completes the life cycle of the smart contract.

### 2.3. Ethereum Running Environment

Blocks, EVM, and smart-contract codes together make up the running environment of Ethereum. The block mining process essentially recognizes legitimate transactions and puts together the corresponding state's transformation into a new block. To achieve this, the miner picks transactions, executes the codes in the smart contract, changes the state, calculates the nonce (proof of work), and attaches the newly mined block onto the previous blockchain. Ethereum accounts are hashed and stored in a data structure called a modified Merkle Patricia tree (Modified MPT) [21] and this Modified MPT's root is hashed and stored in every block. To better understand the smart contract, we explain the Ethereum block structure, as shown in Figure 1.



**Figure 1.** Ethereum block structure.

**Block.** A *block* in Ethereum has three parts: header, transaction list, and ommers list. The header contains three Modified MPT' root nodes (stateRoot, transactionRoot, and receiptsRoot), and other information about the block. Those three Modified MPTs are world state trie, transaction trie, and receipts trie. Within those tries, every account, its state and transactions are stored. There is another Modified MPT, account storage content trie, and its root node is stored in the account state of world state trie. The two following lists are actual transactions and ommers, which are picked by the miner during the mining process. Ommers are blocks that have a parent equal to the present block's parent's parent [21].

**World state trie.** Every leaf node of this trie has a mapping between an account state and its address. Therefore, all accounts, including the external account and contract account are tied together in this trie. Every block has only one world state trie.

**Account storage contents trie.** As we discussed before, the account state has four fields. The account storage content's trie's root node is stored in the field *storage*. Therefore, this trie is not directly stored in the block header, instead of being in the block as part of world state trie. All the contract data is stored in this trie.

**Transaction trie.** This trie is where the hash of all transactions included in a block is stored. Once the block is mined, this trie and transaction list in the block will never be changed.

**Receipts trie.** Every receipt contains results of the execution of transactions, accumulated amount of gas used, logs and status code of the transaction. The serialization of this information combined with keys is stored in the receipts trie.

### 3. Related Work

In recent years, more researchers have studied the security of blockchains and Ethereum smart contracts [13,15–19]. With regards to the security of blockchains in general, Saad et al. [18] and Li et al. [16] surveyed attacks on blockchain and their countermeasures. Similarly, Zhu et al. [17] studied vulnerabilities and defenses on bitcoin blockchain.

### 3.1. Smart-Contracts Vulnerabilities

Luu et al. [13] investigate Ethereum smart-contracts vulnerabilities using their proposed formal verification tool, called Oyente, without explaining defense methods. Dika [11] studies security-analysis tools for the smart contract and provides insights on their effectiveness, accuracy, and consistency. However, his dataset is relatively small and most of the assessed tools were in their beta version at that time. Alharby and Moorsel [12] conduct a systematic mapping study on smart contracts on papers published before 2018 to identify and map research areas related to smart contracts, and discuss codifying, security, privacy, and performance issues. The authors also present a few research gaps in smart-contract research such as the lack of studies on scalability, performance, and deployment issues. Atzei et al. [25] discuss major vulnerabilities and attacks in the context of common programming issues, and present the most cited taxonomy of vulnerabilities of Ethereum smart contracts in three layers: *Solidity*, *EVM*, and *Blockchain*. This categorization is derived from the running structure of Ethereum smart contracts, and most of the vulnerabilities fall into those three classes. Their taxonomy is confined to Ethereum smart-contracts vulnerabilities. In this work, we review the latest advancement and research work published since 2018.

Dingman [26] examines smart-contract codes with the NIST framework and report 49 bugs categorized into 10 classes. Their taxonomy focuses on coding bugs and dives into the technical side of coding. Grishchenko et al. [27] provide a formal definition of the security properties of smart contracts: call integrity, atomicity, independence, and run-time correctness. Di Angelo and Salzer [19] study 27 Ethereum smart-contract security-analysis tools and map them to their purpose as well as to 18 vulnerabilities. Praitheeshan et al. [28] investigate 16 smart-contract vulnerabilities, map them to software security flaws, and discuss and categorize seven security-analysis tools. Huang et al. [29] study common vulnerabilities of the smart contract and examine various tools developed to mitigate risks induced by those vulnerabilities. They categorized those methods into four phases of the software development cycle: design, test, implementation and audit. Sayeed et al. [30] focus on application bugs in the smart contract and discuss seven vulnerability types and 10 security-analysis tools for smart contracts. Durieux et al. [31] evaluate and compare nine smart-contract security-analysis tools on a large dataset of Ethereum smart contracts. Tantikul and Ngamsuriyaroj [32] compare Ethereum smart-contracts vulnerability-detection tools. They perform correlation analysis on detected vulnerabilities in the real world. Chen et al. [14] conduct a thorough survey on Ethereum-system vulnerabilities and defenses. They identify 40 vulnerabilities, 29 attacks, and 51 defenses. The authors map attacks to the vulnerabilities that caused them. However, they do not cover much about detection tools. Tang et al. [15] review Ethereum smart-contract vulnerabilities-detection tools in three categories: static analysis, dynamic analysis, and formal analysis. They consider 15 different security vulnerabilities and present related detection tools. They suggest to use machine-learning methods to analyze smart contracts. They discuss only 15 security vulnerabilities and miss several other important vulnerabilities. Rameder [33] presents a comprehensive systematic literature and tool review of the relevant published studies in the field of Ethereum smart-contract vulnerabilities, detection methods and analysis tools. The author also provides a classification of smart-contract vulnerabilities and a taxonomy of analysis tools. However, countermeasures and tool comparison with regards to accuracy and efficiency, as well as mapping vulnerabilities to CWEs, is missing.

The previous surveys do not cover all aspects of Ethereum smart-contract security vulnerabilities, countermeasures, and tool support and may miss details about tools or defense mechanisms. For instance, while [14,33] conduct an extensive survey, they do not provide tools comparison with regards to their accuracy and efficiency, nor map vulnerabilities to CWE indices. A very recent work [34] conducts a systematic review of vulnerabilities in Ethereum smart contracts, which has a major overlap with our study. While the authors study the same topic as ours and cover more vulnerabilities, they do not consider comparing tools with regards to their accuracy and efficiency.



Table 1 presents a comparison between our study and the existing surveys on smart-contract vulnerabilities. In this work, we aim to complement the above-mentioned studies by considering recent literature and comparing the accuracy and efficiency of some of the existing tools.

**Table 1.** Comparison between the existing surveys on smart contract vulnerabilities and our work.

Survey	Year	Vulnerabilities	Tools	Attacks	Countermeasures	Ethereum Specific
<b>Our Work</b>	2022	✓	✓	✓	✓	✓
Luu et al. [13]	2016	✓		✓		✓
Atzei et al. [25]	2017	✓			✓	
Dika [11]	2017	✓	✓	✓	✓	✓
Alharby and Moorsel [12]	2017	✓			✓	
Grishchenko et al. [27]	2018	✓				✓
Di Angelo and Salzer [19]	2019		✓			✓
Saad et al. [18]	2019			✓	✓	✓
Dingman [26]	2019	✓				✓
Praitheeshan et al. [28]	2019	✓	✓	✓	✓	✓
Huang et al. [29]	2019	✓	✓	✓	✓	
Li et al. [16]	2020	✓		✓	✓	
Zhu et al. [17]	2020	✓		✓	✓	
Sayeed et al. [30]	2020	✓	✓	✓		
Durieux et al. [31]	2020		✓			
Tantikul and Ngamsuriyaroj [32]	2020	✓	✓	✓		✓
Chen et al. [14]	2020	✓		✓	✓	✓
Tang et al. [15]	2021	✓	✓			
Rameder [33]	2021	✓	✓			✓
Kushwaha et al. [34]	2022	✓	✓	✓	✓	✓

### 3.2. Security-Analysis Tools and Methods

Recently, Ghaleb and Pattabiraman [35] propose an automated tool called SolidiFI to evaluate several static analysis tools, including Oyente [13], Securify [36], Mythril [37], SmartCheck [38], Manticore [39] and Slither [40]. SolidiFI injects seven types of bugs into smart contracts, then security-analysis tools are used to detect those bugs. The result reflects the effectiveness of those security tools. The bugs focused are *Timestamp dependency*, *Unhandled exceptions*, *Integer Overflow/Underflow*, *tx.origin*, *re-entrancy*, *unchecked send*, and *transaction order dependence*. SolidiFI can evaluate other static analysis tools, however, it does not reveal evaluation for dynamic analysis and formal verification tools.

Following the survey approach of Praitheeshan et al. [28], we study smart-contract security-analysis methods in the three categories of static, dynamic, and formal verification, for newly proposed security-analysis tools.

#### 3.2.1. Static Analysis

Grech et al. [41] use static analysis to detect *gas-related* vulnerabilities. They decompile the EVM bytecode using Vandal [42] and analyze the output of Vandal to detect patterns such as loops, induction variables and data flow. Chen et al. [43] present a symbolic execution-based tool to detect eight contract defects. It disassembles the smart-contract bytecodes into opcodes and symbolically executes instructions while monitoring patterns such as money call, loop, or payable function. Wang et al. [44] applies machine

learning to detect six vulnerabilities: *integer overflow/underflow*, *transaction-ordering Dependence*, *stack limits*, *timestamp dependency* and *re-entrancy*. Torres et al. develop a tool called OSIRIS that combines symbolic analysis and taint analysis to detect *integer-focused bugs* [45]. Wang et al. [46] focus on detecting *non-deterministic* bugs in smart contracts by analyzing EVM bytecodes and checking flagged global and local variables. Tikhomirov et al. implement a tool called SmartCheck to detect vulnerabilities in Ethereum smart contracts [38], which generates an XML parse tree as an intermediate representation (IR) and uses XPath queries on the IR to track vulnerability patterns. Wang et al. [47] build a tool called Artemis based on the Oyente framework [13] and extend it to detect vulnerabilities such as *Ether lost in transfer* and *delegated call*. Zhang et al. present MPro [48], which is based on Slither [40] and Mythril [37], which optimizes the symbolic execution.

### 3.2.2. Dynamic Analysis

Jiang et al. analyze the ABI specification of smart-contracts functions and generate fuzzing inputs [49]. Ashraf et al. [50] build a fuzzer called GasFuzzer based on Contract-Fuzzer [49], which focuses on the gas consumption of executions and *gas-related vulnerabilities*. He et al. [51] run symbolic execution of real-world contracts to generate thousands of sequences of transactions as their ILF Fuzzer. Nguyen et al. [52] propose an adaptive fuzzer, sFuzz, for smart-contract vulnerabilities detection, which generates many transactions that call functions in the contract, monitors the execution of transactions and collects feedback from the execution. Rodler et al. extend EVMs by implementing a new Ethereum client based on the widely adopted client: go-ethereum [53]. They add a taint engine and attack detector to monitor and detect the run-time state of EVM for re-entrancy vulnerability [54]. Simulator environments [55] could also be used to dynamically execute smart-contract vulnerability detectors based on known patterns or resource usages.

### 3.2.3. Formal Verification

Murray and Anisi conduct a survey on formal verification methods on smart contracts [56]. Their research shows that variants of model checking or theorem-proving methods are successful on simple contracts, not complex contracts or advanced contract syntax. Garfatta et al. [57] compare and discuss a number of tools including FSolidM [58], VeriSolid [59], ZEUS [60], OYENTE [13], and OSIRIS [45], and show that a limited number of vulnerabilities, such as arithmetic bugs, TSD, TOD, re-entrancy, and self destruction, are covered by tools using formal verification methods.

## 4. Smart-Contracts Vulnerabilities and Countermeasures

Normally deemed cryptographically secured, immutable and anonymous, blockchain technology and applications built on it are not necessarily secured. Security issues, vulnerabilities and attacks come up with the broader adoption of smart contracts. The DAO attacks [10] and the Parity Wallet hack [61] are the most notorious attacks that exploited smart-contract vulnerabilities. In this section, we study the vulnerabilities in Ethereum smart contracts written in Solidity to answer RQ1 (vulnerabilities) and RQ2 (countermeasures to mitigate). We first present our taxonomy of smart-contract vulnerabilities in Section 4.1 and then explain each vulnerability and its countermeasures in Section 4.2.

### 4.1. The Proposed Taxonomy

We adapt the three-layer structure taxonomy of [25] and present our updated taxonomy of smart-contract security in Table 2 by categorizing vulnerabilities that are not covered in [25] and omitting some which have been addressed such as stack size limit. The Smart Contract Weakness Classification Registry [62] gathers smart-contract weaknesses and maps them to Common Weakness Enumeration (CWE) [63]. We also add known CWE identifiers to the smart-contract vulnerabilities identified in Table 2 to help correlate our taxonomy with external taxonomies. Note that for vulnerabilities “External contract

referencing”, “Short address/parameter issues”, and “Freezing Ether”, the CWE index was not included in the registry website, and we mapped it by browsing the CWE website.

**Table 2.** Our taxonomy of smart-contract vulnerabilities.

Level	Vulnerability	CWE Index	Real-World Attack
Solidity	Re-entrancy	CWE-841: Improper Enforcement of Behavioral Workflow	The DAO Attack [10]
	Arithmetic issues	CWE-682: Incorrect Calculation	PoWHcoin attack [64]
	Delegatecall to insecure contracts	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	Parity Wallet(Second Hack) [65]
	Selfdestruct	CWE-284: Improper Access Control	Parity Library bug [66]
	Tx.origin	CWE-477: Use of Obsolete Function	-
	Mishandled exceptions	CWE-252: Unchecked Return Value	KingofTheEther attack [67]
	Default visibility	CWE-710: Improper Adherence to Coding Standards	Parity Wallet(First Hack) [68]
	External contract referencing	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	Honey Pot [69]
EVM	Short address/parameter issues	CWE-88: Improper Neutralization of Argument Delimiters in a Command ('Argument Injection')	-
	Freezing Ether	CWE-17: Code Development (Specification, Design, and Implementation)	-
Blockchain	Transaction order dependence	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	Attack on Bancor [70]
	Generating randomness	CWE-330: Use of Insufficiently Random Values	PRNG contract [71]
	Timestamp dependence	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	GovernMental attack [72]

## 4.2. Vulnerabilities and Countermeasures

### 4.2.1. Re-Entrancy

Re-entrancy describes a situation when a contract *A* calls contract *B*, which could call *A* back and execute *A*'s call again. The *fallback mechanism* of Solidity causes this situation. The `fallback` function will be executed when calls from other contracts cannot find a matching function. When the caller uses the `call` function without giving any function signature, callee's `fallback` function will be activated. This function could call the caller's function to re-enter the caller. This mechanism could force unexpected and uncontrolled transfers of Ether in some circumstances.

In the following code excerpt shown in Listing 1 (modified based on [73]), `call` is used to send Ether to the caller. In the beginning, the attacker calls the `withdraw()` function in contract *A* to ask for a one-Ether withdrawal. Contract *A* uses `call` to send the Ether and does not attach any function signature. Contract attacker's `fallback` function responds to this call by calling *A*'s `withdraw` function again. The second call is regarded as a “nested” call inside the previous withdrawal call because the previous one has not finished yet. This is the time that a recursive call begins to form. Normally, the caller will receive



one Ether, and the caller's balance will be deducted as the execution of the instruction `balances[msg.sender] -= _amount;`. However, in a *re-entrancy situation*, this instruction will never be executed because the `call` function above this line will lead to recursively calling the `withdraw()` function until the total balance of Contract A is less than one Ether or the gas limit is reached.

**Listing 1.** Re-entrancy vulnerability.

```
contract A {
    mapping (address => uint) public balances;
    function withdraw(uint _amount) public {
        msg.sender.call{value: _amount}('');
        balances[msg.sender] -= _amount;
    }
    /*Other codes*/
}

contract Attacker {
    A public a;
    fallback() external payable {
        a.withdraw(1 ether);
    }
    function attack(){
        a.withdraw(1 ether);
    }
}
```

**Countermeasures.** There are three techniques to prevent re-entrancy. The first is to restrain using `call` function whenever possible. The `transfer` function could be used to send Ether to other accounts. This function only sends 2300 gas with external calls, so the called contract will not have enough gas to re-enter the caller contract. The second is to add a “lock”, which is usually a state variable that shows the state of external calls and permits the external call only if the state is right [74]. The third is to make all logic changes before the `call` function is executed. For example, in the code snippet in Listing 1 moving `balances[msg.sender] -= _amount` to before `msg.sender.call{value: _amount}('')`.

#### 4.2.2. Arithmetic Issues

In Solidity, integer type has a specific range. During the arithmetic operations on variables, their values may go beyond the upper or lower bound. If that happens, the value will warp to the other side of the bound. For example, if the real value is larger than the upper bound, the actual value will be the real value less the upper bound. In earlier versions of Solidity, this abnormality does not trigger any alarm. Attackers could increase or decrease specific integer values to trick the smart contract into unwanted behaviour. Solidity does not support floating points yet. The representation of floating numbers in smart contracts must be dealt with integer types. For example, in the design of ERC20 (a token standard), decimal is used to represent the number of digits after the decimal point. The division in Solidity always rounds down, so the precision is only to its nearest lower integer. The loss of information after the decimal points could cause severe problems when higher precision is needed.

**Countermeasures.** Since arithmetic vulnerability lies in Solidity, using a well-designed, audited library such as SafeMath [75] provided by Openzeppelin [76] instead of built-in arithmetic operations will mitigate the risk.

#### 4.2.3. Delegatecall to Insecure Contracts

Delegatecall is a special function in Solidity. It is the same as the `call` function with one major difference. The `delegatecall` re-uses codes of the called contract and executes them in the caller's context. For example, in the following code excerpt in Listing 2 (modified based on [77]), `msg.sender`, `msg.data` in the contract `VulnerableContract` will be the caller's address and data, and the state variables will use the caller contract's. Once the contract

InsecureContract is called via `delegatecall`, the function `doSomething` is executed in the caller's context in which the caller's state variable `owner` in the contract `VulnerableContract` will have an unexpected modification. The `VulnerableContract` block does not provide any way to change its ownership once it is constructed. However, when the attacker calls this contract at a certain point of execution, it will `delegatecall` `InsecureContract` in which the state variable `owner` is modified to the new value of `msg.sender`, which is the address of the attacker. In other words, `delegatecall` gives other contracts, such as `InsecureContract`, permission to alter its own state variables, such as `address public owner`. This is because the `delegatecall` function preserves context.

**Listing 2.** Delegatecall to insecure contracts.

```
contract VulnerableContract {
    address public owner;
    InsecureContract public c;
    constructor(InsecureContract _c) public {
        owner = msg.sender;
        c = InsecureContract(_c);
    }
    /*Other codes...*/
    c.delegatecall(abi.encodeWithSignature('doSomething()'));
    /*Other codes...*/
}

contract InsecureContract {
    address public owner;
    function doSomething() public {
        /*Other codes...*/
        owner = msg.sender;
        /*Other codes...*/
    }
}
```

**Countermeasures.** Since `DELEGATECALL` exploits the context change of smart contracts, the first thing to do is checking the possible context of contracts and their calling libraries (or other contracts). Solidity also provides keyword `Library` to restrict aligning state variable slots with a caller's state variables [78]. The Solidity library cannot have its state variable, so there will be no context switching and unauthorized change of caller's state variables.

#### 4.2.4. Selfdestruct

The `selfdestruct` operation in Solidity provides a way to remove contracts from the following blocks (it is still part of the historical blocks in the blockchain). This operation is dangerous because if senders are not all updated with the new contract's address, some will still send Ether to the destructed contract, which will cause the loss of Ether. Moreover, this process will forcefully send the remaining Ether of the contract to a designated address. Malicious adversaries could exploit this behaviour to *force-send Ether* to some contracts. The victim contract's `this.balance` will be manipulated with this force-send, especially if `this.balance` is used as conditions for certain operations.

**Countermeasures.** To avoid an unexpected `selfdestruct` call, try the best practice of using caution when making external calls [79]. In the meantime, the contract's logic should avoid depending on the value of `this.balance`, because it can be forcibly modified by attackers.

#### 4.2.5. Freezing Ether

This vulnerability is also known as greedy contract [43], and locked money [38]. A smart contract is designed to be able to receive and send Ether. A freezing-Ether situation is a contract that could only receive Ether but has no means to send Ether out. A contract can be greedy and the Ether sent to its address is frozen if the contract does not define any withdraw functions.

**Countermeasures.** It is a good practice to avoid this situation at the smart-contract designing phase. Those contracts that could receive Ethers should have functions to withdraw Ethers [43].

#### 4.2.6. Randomness Generation

Generating random numbers is a problem for many programming languages. The pseudo-randomness generation process leverages certain secret seeds to achieve a level of randomness. However, seeds cannot be privately stored on the chain in a smart contract in Solidity because everything on the blockchain is visible to its participants. The current practice is using block-related information such as `block.timestamp` or `block.hash`. If historical blocks' timestamp or hash is used, attackers can use the same random number-generation process to obtain the same result because historical blocks never change; if using the future blocks, the process may be susceptible to malicious miners who can intentionally choose transactions and their execution orders.

**Countermeasures.** Block-related information should not be used as an entropy source for randomness. The source should be outside of the blockchain environment. Other proposals are a crowd commit to the random number and reveal process (commit-reveal [80]) or RANDAO [81].

#### 4.2.7. tx.origin

The `tx.origin` is a unique global variable in Solidity. It stores the original caller's address of a transaction, unlike `msg.sender`, which is the immediate caller. Therefore, `tx.origin` is always an external account's address. If this variable is used as an authorization parameter, the identity of the actual owner of a smart contract could be exploited. For instance, if `require(tx.origin == owner)`; is invoked by the attacker's contract, which is called by the real owner, the attacker could go on to execute code that follows.

**Countermeasures.** Attackers could run their code under the name of caller's `tx.origin`, so never use `tx.origin` in identity verification or authentication [79,82].

#### 4.2.8. Mishandled Exceptions

In Solidity, the `require`, `assert`, `try/catch` could be utilized to verify all kinds of information to ensure that the smart contract behaves as it is designed to. However, some low-level functions such as `address.send()`, `call()`, `delegatecall()`, `staticcall()` find another way when an error comes up. Normally, when an exception is found, the transactions are reverted and gas is consumed, but if low-level functions encounter errors (e.g., the call stack is depleted), they will return a `false` value without any exceptions. Any transactions executed before those functions will not be reverted, and gas is spent. In other words, when using low-level functions, a `false` return value should be noted and properly handled; otherwise, the security of the smart contract will be affected.

**Countermeasures.** As suggested by Solidity Documentation [83], low-level functions should not be used whenever possible. The `transfer` function can be used to transfer Ether to other accounts. If those low-level functions must be used, it is better to check every return value of those functions and handle the false ones.

#### 4.2.9. Timestamp Dependence

Blocks in Ethereum are created in the mining process in which the miner has a certain degree of arbitrariness to decide the timestamp of the whole block. The time range used to be 900 s, but it was narrowed down to a few seconds after an upgrade [84]. All the transactions recorded on the block have the same timestamp. Since the miner can decide timestamps on transactions, certain applications that function correctly under time constraints are susceptible to malicious miners. The timestamp is also used as a seed when generating random numbers. Attackers may manipulate the timestamp of a block to trigger expected behaviours from smart contracts [28].

**Countermeasures.** Antonopoulos and Wood [74] mention that using `block.number` is a better choice than `block.timestamp`. As block generating speed is usually 10 s, there will be 60,480 blocks, approximately, in one week. By specifying a future block number, a time estimation comes with this future `block.number`.

#### 4.2.10. Transaction Order Dependence

Since transactions change the world state from one to another, the world state or state of contracts depends on the transaction execution order. The miner arbitrarily decides the execution order, and this non-deterministic feature of the execution order makes it difficult to forecast the state before transactions are submitted. In a real-world situation where a transaction depends on the state of the contract, a different order may cause severe problems such as buying or selling items at unexpected prices [28].

**Countermeasures.** Transaction order is affected by accounts that pay the higher gas price or miners who decide the order. If an upper bound is put on the gas price, a proportion of transactions will not be exploited by attackers willing to pay more gas prices.

#### 4.2.11. Default Visibility

Solidity has visibility specifiers, `public`, `private`, `external`, and `internal`, that control functions and variables' visibility from outside of the contract and if `public` is not set as the default. If a developer does not specify an internally-used function as `private`, that function could be called from outside the contract, which will lead to unexpected operation escalation.

**Countermeasures.** It is a good practice to always determine specifiers and Solidity also generates warnings for functions without specifiers while compiling.

#### 4.2.12. External Contract Referencing

Smart contracts need to re-use other contracts' codes sometimes by referencing their addresses. For example, a developer could initiate an outside library in his contract's constructor. The library's address will be provided during deployment. This seems legitimate and safe from code audit. However, if given a wrong address in the deployment, a security hole is created because a malicious contract (honey pot) may be called from that wrong address.

**Countermeasures.** One could hard code the external addresses into contract's codes if they are public or use `new` keyword to create contracts instead of deployment input.

#### 4.2.13. Short Address/Parameter Issues

The parameters of smart-contract functions are encoded before passing to it. The encoded parameter is 32 bytes, and the EVM will concatenate all the encoded parameters together and pad 0 at the end if needed [83]. If the first parameter does not have enough length, e.g., 30 bytes, this will cause a 2-byte left shift after the EVM padding. The shift and padding will increase the value of the second parameter, which could cause severe issues if the second parameter denotes the number of tokens or Ether.

**Countermeasures.** To address this vulnerability, parameters sent to the smart contracts on Ethereum should be validated first. This is a best practice for third-party applications that interact with smart contracts. For smart-contract developers, parameter order matters since the padding only happens at the end of concatenated parameters.

### 4.3. Vulnerability Causes

Based on previous work [11,13,22,25,85], we summarize several causes for smart-contract vulnerabilities.

- **Visibility:** Ethereum records all of its validated historical transactions in *world state* (refer to Section 2), which is visible to the whole network. This visibility may cause some problems when the smart-contract owner wants to keep it from the public due to privacy concerns. Prior studies have shown that inspecting accumulated transactions' statistic characteristics and leveraging analysis graph structures could reveal valuable

and practical information, given users are granted anonymity with pseudonymous public keys [86,87].

- **Opacity:** On the other hand, the opacity of smart contracts poses another concern. Live contracts need rigid inspection and source-code audits since they manage millions of USD. However, 77% of running smart contracts are opaque and hold USD 3 billion [88].
- **Immutability:** Smart contracts cannot be modified once deployed on Ethereum. This immutability perpetuates problematic contracts on the chain. Workarounds such as updating contracts or addresses at the user side could solve the problem to a certain degree, however, they introduce other security concerns.
- **Automated Execution:** A control flow transfer (CFT) between two instructions *A* and *B* means that after *A*'s execution, *B* is executed immediately. When smart contracts interact with each other, those CFTs are not always under control. In the uncontrolled-function call-permission cases, the self-elevation of a malicious attacker could happen by triggering some high-privilege functions [85].
- **Mining:** Which transactions to be processed depends on the miner's choice. The *deterministic* result of a single transaction's execution no longer exists when multiple transactions are involved. Their execution order may vary based on the miner's choice. This uncertainty severely affects *order-sensitive transactions*. The final result of a series of transactions might reach vastly different outcomes.
- **EVM:** EVM structure provides a stand-alone, isolated running-time environment (sandbox) for smart contracts.  
EVM utilizes an oracle to handle the incoming real-world inputs. An oracle is a concept that bridges blockchain and the real world and acts as APIs on chain, which can be accessed by the smart contracts to receive feeds from the real world [89]. Centralized oracles introduce several attack vectors and issues to smart contract such as *integrity*, *accuracy*, and *consistency* [90]. However, the decentralized designs of oracle such as Chainlink [91] are trying to solve this problem.
- **Immature Programming Language:** Solidity is an evolving programming language and its widely known vulnerability is the *re-entrancy problem*, which is caused by the fallback mechanism in Solidity. Mishandled exceptions could be a severe problem for smart contracts too.

## 5. Security-Analysis Tools

Existing security-analysis tools for smart contracts mainly apply static or dynamic analysis. Newly proposed tools incorporate other approaches such as deep learning or focus on certain types of vulnerabilities. In order to answer RQ3, in this section, we study smart-contract security-analysis tools and compare nine recently proposed tools that were released after 2017. Our work complements [11], which was carried out in 2017. Compared to our study, [33] covers many more tools. While they consider tools introduced since 2014, we focus on recently proposed tools after 2018 that are specific to Ethereum and are also prevalently used or mentioned in multiple recent studies. Moreover, we did not consider a number of tools that either have no coverage of our set of vulnerabilities or their developers/authors do not report their efficiency or accuracy analysis.

### 5.1. Recent Security-Tools Support

Table 3 shows how each tool covers the most ubiquitous vulnerabilities. Many tools have their own set of vulnerabilities and methods to find them (see Table 4). There are overlapping vulnerabilities that many tools try to detect. We consider reported accuracy above 90% as **high**, below 50% as **low**, and the rest as **medium**. The accuracy works in the realm of each tool's defined vulnerabilities, so this comparison will give readers an impression of how each tool works for their own set of vulnerabilities.



**Table 3.** Tools and vulnerabilities matrix.

Security Tool	Re-Entrancy	Arithmetic Issues	Delegatecall to Insecure Contracts	Selfdestruct	Tx.origin	Mishandled Exceptions	External Contract Referencing	Short Address	Freezing Ether	Transaction Order Dependence	Generating Randomness	Timestamp Dependence
Oyente	✓					✓				✓		✓
Securify	✓		✓			✓		✓	✓	✓		
SmartCheck	✓	✓			✓				✓			✓
DefectChecker	✓					✓			✓	✓		✓
ContractWard	✓	✓								✓		✓
NPChecker	✓									✓		✓
MadMax		✓										
Osiris		✓										
ContractFuzzer	✓		✓	✓		✓			✓			✓
Sereum	✓											
sFuzz	✓	✓	✓	✓		✓			✓			✓

**smartCheck.** This tool is a static analysis tool that uses ANTLR [92] (a parser generator) to translate Solidity source code into an XML parse tree [93] (an XML-based intermediate representation), and checks it against XPath [94] patterns [38].

**DefectChecker.** The defectChecker tool takes bytecodes as input, disassembles them into opcodes, splits the opcodes into several basic blocks and symbolically executes instructions in each block [43]. Then it generates the control flow graph (CFG) and records all stack events. Using CFG and stack events information, it detects three pre-defined features: money call, loop block and payable function. After feature detection, it applies rules to detect eight vulnerabilities: *transaction state dependency*, *DoS under external influence*, *strict balance equality*, *re-entrancy*, *nested call*, *greedy contract*, *unchecked external calls*, and *block info dependency*.

**contractWard.** The contractWard applies supervised learning to find vulnerabilities [44]. It extracts 1619 dimensional bigram features from opcodes using an n-gram algorithm [95] and forms a feature space. Then it labels contracts in training set with six types of vulnerabilities using Oyente [13]. The label is stored in a six-dimension vector (e.g., [1 0 1 0 1 1]) where each bit stands for an existing vulnerability. Based on the feature space and labels of the training set, contractWard uses five classification algorithms to detect vulnerabilities.

**NPChecker.** This tool analyzes the *non-determinism* in the smart-contract execution context and then performs systematic modelling to expose various non-deterministic factors in the contract execution context [46]. Non-deterministic factors are factors that could impact final results to the end-user and make them unforeseeable. Possible factors discussed in NPChecker are *block and transaction state*, *transaction execution scheduling*, and *external callee*. The NPChecker disassembles the EVM bytecode and translates them into LLVM intermediate representation (IR) code [96], recovers the control flow structures and enhances the LLVM IR with function information, identifies state and global variables, and performs information-flow tracking to analyze their influences on the fund's transfer.

**MadMax.** The MadMax tool uses the Vandal decompiler [42] to produce CFG, a three-address code for all operations in the smart contract, and function boundaries [41]. Then it analyzes the three-address-code representation, recognizes concepts such as loops and induction variables, analyzes memory and dynamic data structures, and infers the concept of gas-focused vulnerabilities. This tool only detects *gas-focused vulnerabilities*, such as unbounded mass operations (infinite or nearly infinite loops), external calls that throw out-of-gas exceptions or arithmetic integer overflows, because those vulnerabilities will cause unexpected gas consumption.

**Osiris.** The Osiris combines symbolic analysis and taint analysis technology to find integer bugs in smart contracts [45]. It has three components: symbolic analysis, taint analysis and integer error detection. The symbolic analysis component creates CFG and symbolical executions of every path in the CFG. Then, the taint analysis part checks for taints across the stack, memory and storage, and integer error detection looking for possible integer bugs within the executed instruction.

**contractFuzzer.** The contractFuzzer tool combines static analysis of ABI and bytecodes and fuzzing technology to explore vulnerabilities of smart contracts [49]. It creates an Ethereum test net using offline EVM to monitor the execution of the smart contracts and extract information from the execution process. By analyzing the ABIs and bytecodes, contractFuzzer calculates the function selector (first four bytes of the hash of the function's signature) and maps each ABI function to a set of function selectors used. Then an input generation algorithm is created to fuzz each function based on the information of the previous step. It collects three types of test oracles during the execution of smart contracts: attributes of a contract `call` or `delegatecall`, run-time information about opcodes invoked, and the state of the contract.

**Sereum.** It is a modified Ethereum client based on Geth [97] that focuses on re-entrancy vulnerabilities [54]: *cross-function re-entrancy*, *delegated re-entrancy*, and *create-based re-entrancy*. The cross-function re-entrancy is to re-enter another function in the same contract. The delegated re-entrancy is to re-enter a smart contract via `delegatecall` to an unsafe library that may use `address.call().value()` to call back to the original contract. The create-based re-entrancy utilizes the fact that a newly created contract will have its constructor function executed immediately when the contract is deployed. The constructor is deemed safe and trusted, but it could contain external calls to malicious contracts. If the victim contract creates another contract in function *A* and calls the attacker's contract in the newly created contract's constructor, the attacker's contract could re-enter the victim contract by calling function *A*. Sereum adds two components to the Geth client: taint engine (taints and tracks state variables along with the executions of various functions that detects conditional `JUMP` instructions influenced by a storage variable), and attack detector (creates locks that prohibit further updates for state variables that influence control flows).

**sFuzz.** The sFuzz tool is an adaptive fuzzer that combines the strategy in the AFL fuzzer [98] and other adaptive strategies [52] built on Aleth [99] with implementations of three more components: *runner*, *libfuzzer*, and *liboracles*. The *runner* sets up the test net environment and options of the other two components. Then the contracts are executed on the test net, where transactions are generated based on the analysis of the contract's ABI. The *libfuzzer* selectively generates test cases by implementing a feedback-guided adaptive fuzzing strategy. The *liboracles* monitors the execution of a test case and the corresponding stack events to check for vulnerabilities.

## 5.2. Comparisons

Table 4 summarizes our comparisons and shows that new tools are more static analyzers and are focused on self-defined or widely accepted sub-categories of vulnerabilities. The first two rows are based on [11] that, prior to 2018, studied Oyente [13], Securify [36], a beta version of SmartCheck (that was later published in 2018), and Remix [100], which now uses Securify for safety assurance. The NPChecker defines non-deterministic factors and works on vulnerabilities raised by those factors. The MadMax, Osiris, and Sereum focus on

gas-related, arithmetic, and re-entrancy problems, respectively. We extracted evaluation information from each cited paper for our comparison and report accuracy and efficiency in the Table 4. We explain threats to the validity of our comparison in Section 6.

**Table 4.** A comparison of security-analysis tools. The first two rows were studied before 2018 by Dika in [11]. Note that the accuracy and efficiency of these tools are not really comparable as they rely on self-defined metrics by their authors. We consider reported accuracy above 90% as high, below 50% as low, and the rest as medium.

Tools Name	Methods	Vulnerabilities Covered	Accuracy	Efficiency
Oyente [13]	Static	<b>Vulnerabilities in the taxonomy:</b> Transaction order dependence, Timestamp dependence, Mishandled exceptions, Re-entrancy.	High (94.3%)	29.5 s per execution path
Securify [36]	Static	<b>Vulnerabilities in the taxonomy:</b> Freezing Ether, Re-entrancy, Delegatedcall to insecure contracts, Mishandled exceptions, Transaction order dependency, Short address/parameter issues	Not reported	30 s per contract
SmartCheck [38]	Static	<b>Vulnerabilities in the taxonomy:</b> Re-entrancy, Tx.origin, Freezing Ether, Arithmetic Issues, Timestamp dependence; <b>Self-defined vulnerabilities:</b> Unchecked external calls, DoS by external contract	Low (31.11%)	1.67 per contract
DefectChecker [43]	Static	<b>Vulnerabilities in the taxonomy:</b> Transaction order dependence, Re-entrancy, Freezing Ether, Mishandled exceptions, Timestamp dependence; <b>Self-defined vulnerabilities:</b> DoS, Strict balance equality, Nested call, Unchecked external calls.	Medium (84.16%)	2.42 s per contract
ContractWard [44]	Static	<b>Vulnerabilities in the taxonomy:</b> Arithmetic issues, Transaction order dependency, timestamp dependency, Re-entrancy; <b>Self-defined vulnerabilities:</b> stack size limit (deprecated)	High (96%)	4 s per contract
NPChecker [46]	Static	<b>Vulnerabilities in the taxonomy:</b> Timestamp dependence, Re-entrancy, Transaction order dependence; <b>Self-defined vulnerabilities:</b> external calls and other non-determinism-related smart-contract payment bugs.	High (94%)	351 s per contract
MadMax [41]	Static	<b>Vulnerabilities in the taxonomy:</b> Arithmetic issues; <b>Self-defined vulnerabilities:</b> Unbounded mass operations, Non-isolated external calls.	Medium (81%)	5.9 s per contract
Osiris [45]	Static	<b>Vulnerabilities in the taxonomy:</b> Arithmetic issues; <b>Self-defined vulnerabilities:</b> Division by zero, Modulo zero, Type cast, Signedness.	Not reported	75 s per contract
ContractFuzzer [49]	Dynamic	<b>Vulnerabilities in the taxonomy:</b> Selfdestruct, Mishandled exception, Re-entrancy, Timestamp dependency, Delegatedcall to insecure contracts, Freezing ether.	High (98.91%)	Not reported
Sereum [54]	Dynamic	<b>Self-defined Re-entrancy vulnerabilities:</b> Cross-function reentrancy, Delegated reentrancy and Create-based reentrancy	High (99.4%)	2494.5 ms per block runtime (2277.0 ms on Geth)
sFuzz [52]	Dynamic	<b>Vulnerabilities in the taxonomy:</b> Selfdestruct, Mishandled exceptions, Re-entrancy, Timestamp dependency, Delegatecall to insecure contracts, Arithmetic Issues, freezing ether	High (90.56%)	Generates 208 test cases per second on average.

**Accuracy:** Based on their own understanding of the smart-contract vulnerabilities and causes, each tool has its own metrics to determine true positives and false positives, and the statistics are strongly related to the covered vulnerabilities. The accuracy numbers for Oyente and Securify are calculated based on [13] and Figure 8 in [11], respectively.

**Efficiency:** The efficiency is evaluated based on each tool's approach. It is based on the process time of each contract for the static tools, and it is based on the specific mechanism of each dynamic tool. For Oyente and Securify, no efficiency data is reported either in [11] or in their own proposals. Sereum is a modified Geth [97] client (a client that interacts

with Ethereum blockchain) and replays the execution of each transaction in the blockchain; thus, its efficiency is compared with Geth client running performance. In addition, sFuzz's efficiency is evaluated on how fast it can generate fuzzing inputs.

## 6. Discussion

### 6.1. Findings

Our findings regarding RQ1 (vulnerabilities) indicate that a uniform set of smart-contract vulnerability definitions does not exist in research work and bugs pertaining to the same mechanisms sometimes appear with different names. For example, the *mishandled exception* vulnerability has been referred to as *exception disorder* in [25], *unchecked-send bug* in [11], *unhandled exception* in [101], and *unchecked external calls* in [43,102]. This inconsistency makes it difficult to identify, categorize, and analyze vulnerabilities, as well as challenging to compare security-analysis tools. The Smart Contract Weakness Classification Registry [62] gathers smart-contract weaknesses and maps them to Common Weakness Enumeration (CWE) [63]. The community also has Ethereum Improvement Proposal [103], which describes standards for the platform, protocol and contract, and guidelines for Ethereum smart-contract best practices [79] to approach a better security level for smart contracts and their run-time environment. For RQ2 (countermeasures to mitigate), we explained some safeguarding approaches and best practices. However, as technology improves new vulnerabilities may emerge.

As far as RQ3 (tool support) is concerned, the results indicate that SmartCheck, DefectChecker, contractWard and sFuzz tools are better choices in terms of more coverage of vulnerabilities; however, tools such as NPChecker, MadMax, Osiris, and Sereum target some specific categories of vulnerabilities if required. While contractWard has good accuracy and fast process time, it can only detect specifically pre-defined vulnerabilities. The NPChecker has a slower processing time, but it may find new patterns for each vulnerability.

Our results complement [35] and only share the SmartCheck tool. However, their data is SolidiFI-centric and thus there is no direct comparison between their accuracy and efficiency with ours. We found that there is inconsistency in the definition of various vulnerabilities in different research work. This poses challenges to creating unified metrics or benchmarks for comparisons among security-analysis tools.

### 6.2. Threats to Validity

A threat to the external validity of our study is with regards to the comparison of the tools in terms of their accuracy. We cannot simply compare those tools' accuracy because each tool has its own dataset and metric to define true positives and false positives. These metrics are derived from different understandings and scopes of vulnerabilities defined and hence it is not easy to include all the vulnerabilities in one dataset and experiment tools based on that. However, reported numbers could reflect the efficacy of the tool in detecting target vulnerabilities. Another external validity threat is regarding the generalization of the studied tools. We acknowledge that our set of tools is not comprehensive, and there exist other tools, particularly very recent ones, that we might have missed in our comparison. However, we believe that the studied tools represent recently proposed tools specific to Ethereum and are also prevalently used or mentioned in multiple recent studies. Moreover, we did not consider several tools that either had no coverage of our set of vulnerabilities or their developers/authors did not report their efficiency or accuracy analysis. Our comparison in Section 5.2 is based on what the authors claim in the papers. However, tools may evolve, and research papers may not have up-to-date lists of vulnerabilities detected by the tool.

## 7. Conclusions and Future Work

We surveyed 13 vulnerabilities in Ethereum smart contracts and their countermeasures, and investigated nine security-analysis tools. The inconsistencies in definitions of various vulnerabilities in different research work make it challenging to create a unified metric or

benchmark for comparison among security-analysis tools. Consistent naming conventions and widely accepted definitions of vulnerabilities are highly helpful for further research and comparisons. More research efforts are needed to expand this work, and new security-analysis techniques are needed to further enhance smart contracts' security. Another future extension of this work is studying vulnerabilities related to blockchain platforms other than Ethereum such as Hyperledger Fabric, Hyperledger Sawtooth, EOS, Tezos, Solana, Corda, NEO, and Cardano.

**Author Contributions:** Conceptualization, H.Z., A.M.F. and A.M.; methodology, H.Z. and A.M.F.; validation, H.Z., A.M.F. and A.M.; formal analysis, H.Z. and A.M.F.; investigation, H.Z. and A.M.F.; writing—original draft preparation, H.Z., A.M.F. and A.M.; writing—review and editing, H.Z., A.M.F. and A.M.; visualization, H.Z. and A.M.F.; supervision, A.M.F. and A.M.; project administration, A.M.F. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

- Schär, F. Decentralized finance: On blockchain-and smart contract-based financial markets. *SSRN* **2020**, *2020*, 3571335. [CrossRef]
- Bartoletti, M.; Pompianu, L. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, Sliema, Malta, 3–7 April 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 494–509.
- Ethereum. Available online: <http://www.ethereum.org/> (accessed on 19 January 2022).
- Foundation, E. Available online: <https://docs.soliditylang.org/en/v0.8.2/index.html> (accessed on 19 January 2022).
- Coin Market Cap. 2013. Available online: <https://coinmarketcap.com/currencies/ethereum/> (accessed on 19 January 2022).
- DefiPulse. 2019. Available online: <https://defipulse.com/> (accessed on 19 January 2022).
- CoinMarketCap. 2013. Available online: <https://coinmarketcap.com/> (accessed on 19 January 2022).
- Nakamoto, S.; Bitcoin, A. A Peer-to-Peer Electronic Cash System. 2008. Available online: <https://bitcoin.org/bitcoin.pdf> (accessed on 19 January 2022).
- Patron, T. What's the Big Idea Behind Ethereum's World Computer? 2016. Available online: <https://www.coindesk.com/whats-big-idea-behind-ethereums-world-computer> (accessed on 19 January 2022).
- Mehar, M.I.; Shier, C.L.; Giambattista, A.; Gong, E.; Fletcher, G.; Sanayhie, R.; Kim, H.M.; Laskowski, M. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *J. Cases Inf. Technol. (JCIT)* **2019**, *21*, 19–32. [CrossRef]
- Dika, A. Ethereum Smart Contracts: Security Vulnerabilities and Security Tools. Master's Thesis, Norwegian University of Science and Technology, Trondheim, Norway, December 2017.
- Alharby, M.; Van Moorsel, A. Blockchain-based smart contracts: A systematic mapping study. *arXiv* **2017**, arXiv:1710.06372.
- 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 (CCS'16)*, Vienna, Austria, 24–28 October 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 254–269. [CrossRef]
- Chen, H.; Pendleton, M.; Njilla, L.; Xu, S. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Comput. Surv.* **2020**, *53*, 67. [CrossRef]
- Tang, X.; Zhou, K.; Cheng, J.; Li, H.; Yuan, Y. The Vulnerabilities in Smart Contracts: A Survey. In *Advances in Artificial Intelligence and Security*; Sun, X., Zhang, X., Xia, Z., Bertino, E., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 177–190.
- Li, X.; Jiang, P.; Chen, T.; Luo, X.; Wen, Q. A survey on the security of blockchain systems. *Future Gener. Comput. Syst.* **2020**, *107*, 841–853. [CrossRef]
- Zhu, L.H.; Zheng, B.K.; Shen, M.; Gao, F.; Li, H.Y.; Shi, K.X. Data Security and Privacy in Bitcoin System: A Survey. *J. Comput. Sci. Technol.* **2020**, *35*, 843–862. [CrossRef]
- Saad, M.; Spaulding, J.; Njilla, L.; Kamhoua, C.; Shetty, S.; Nyang, D.; Mohaisen, A. Exploring the Attack Surface of Blockchain: A Systematic Overview. *arXiv* **2019**, arXiv:1904.03487.
- di Angelo, M.; Salzer, G. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *Proceedings of the 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, Newark, CA, USA, 4–9 April 2019; pp. 69–78. [CrossRef]
- Buterin, V. *A Next-Generation Smart Contract and Decentralized Application Platform*; Ethereum Foundation: Zug, Switzerland, 2013.
- Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Proj. Yellow Pap.* **2014**, *151*, 1–32.



22. Zheng, Z.; Xie, S.; Dai, H.N.; Chen, W.; Chen, X.; Weng, J.; Imran, M. An overview on smart contracts: Challenges, advances and platforms. *Future Gener. Comput. Syst.* **2020**, *105*, 475–491. [CrossRef]
23. Vbutterin. Serpent. Available online: <https://github.com/ethereum/serpent> (accessed on 19 January 2022).
24. CornellBlockchain. Bamboo. Available online: <https://github.com/CornellBlockchain/bamboo> (accessed on 19 January 2022).
25. Atzei, N.; Bartoletti, M.; Cimoli, T. A survey of attacks on ethereum smart contracts (sok). In Proceedings of the International Conference on Principles of Security and Trust, Uppsala, Sweden, 24–25 April 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 164–186.
26. Dingman, W.; Cohen, A.; Ferrara, N.; Lynch, A.; Jasinski, P.; Black, P.E.; Deng, L. Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework. *Int. J. Netw. Distrib. Comput.* **2019**, *7*, 121–132. [CrossRef]
27. Grishchenko, I.; Maffei, M.; Schneidewind, C. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust*; Bauer, L., Küsters, R., Eds.; Springer: Cham, Switzerland, 2018; pp. 243–269.
28. Praitheeshan, P.; Pan, L.; Yu, J.; Liu, J.; Doss, R. Security analysis methods on Ethereum smart contract vulnerabilities: A survey. *arXiv* **2019**, arXiv:1908.08605.
29. Huang, Y.; Bian, Y.; Li, R.; Zhao, J.L.; Shi, P. Smart contract security: A software lifecycle perspective. *IEEE Access* **2019**, *7*, 150184–150202. [CrossRef]
30. Sayeed, S.; Marco-Gisbert, H.; Caira, T. Smart contract: Attacks and protections. *IEEE Access* **2020**, *8*, 24416–24427. [CrossRef]
31. Durieux, T.; Ferreira, J.A.F.; Abreu, R.; Cruz, P. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Korea 27 June–19 July 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 530–541. [CrossRef]
32. Tantikul, P.; Ngamsuriyaroj, S. Exploring Vulnerabilities in Solidity Smart Contract. In Proceedings of the 6th International Conference on Information Systems Security and Privacy ICISPP, Valletta, Malta, 25–27 February 2020; pp. 317–324.
33. Rameder, H. Systematic Review of Ethereum Smart Contract Security Vulnerabilities, Analysis Methods and Tools. Ph.D. Thesis, Manipl University, Wien, Austria, 2021.
34. Kushwaha, S.S.; Joshi, S.; Singh, D.; Kaur, M.; Lee, H.N. Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. *IEEE Access* **2022**, *10*, 6605–6621. [CrossRef]
35. Ghaleb, A.; Pattabiraman, K. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Online, 18–22 July 2020; pp. 415–427.
36. Tsankov, P.; Dan, A.; Drachler-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, Toronto, ON, Canada, 15–19 October 2018; pp. 67–82.
37. ConsenSys. Mythril. 2021. Available online: <https://github.com/ConsenSys/mythril> (accessed on 19 January 2022).
38. Tikhomirov, S.; Voskresenskaya, E.; Ivanitskiy, I.; Takhaviev, R.; Marchenko, E.; Alexandrov, Y. Smartcheck: Static analysis of ethereum smart contracts. In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, Gothenburg, Sweden, 27 May–3 June 2018; pp. 9–16.
39. Mossberg, M.; Manzano, F.; Hennenfent, E.; Groce, A.; Grieco, G.; Feist, J.; Brunson, T.; Dinaburg, A. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1186–1189.
40. Feist, J.; Grieco, G.; Groce, A. Slither: A static analysis framework for smart contracts. In Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Montreal, QC, Canada, 27 May 2019; pp. 8–15.
41. Grech, N.; Kong, M.; Jurisevic, A.; Brent, L.; Scholz, B.; Smaragdakis, Y. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.* **2018**, *2*, 1–27. [CrossRef]
42. Brent, L.; Jurisevic, A.; Kong, M.; Liu, E.; Gauthier, F.; Gramoli, V.; Holz, R.; Scholz, B. Vandal: A scalable security analysis framework for smart contracts. *arXiv* **2018**, arXiv:1809.03981.
43. Chen, J.; Xia, X.; Lo, D.; Grundy, J.; Luo, X.; Chen, T. DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. *IEEE Trans. Softw. Eng.* **2021**, *1*, 1. [CrossRef]
44. Wang, W.; Song, J.; Xu, G.; Li, Y.; Wang, H.; Su, C. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Trans. Netw. Sci. Eng.* **2020**, *8*, 1133–1144. [CrossRef]
45. Torres, C.F.; Schütte, J.; State, R. Osiris: Hunting for integer bugs in ethereum smart contracts. In Proceedings of the 34th Annual Computer Security Applications Conference, San Juan, PR, USA, 3–7 December 2018; pp. 664–676.
46. Wang, S.; Zhang, C.; Su, Z. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [CrossRef]
47. Wang, A.; Wang, H.; Jiang, B.; Chan, W. Artemis: An improved smart contract verification tool for vulnerability detection. In Proceedings of the 2020 7th International Conference on Dependable Systems and Their Applications (DSA), Xi'an, China, 28–29 November 2020; pp. 173–181.
48. Zhang, W.; Banescu, S.; Pasos, L.; Stewart, S.; Ganesh, V. MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract. In Proceedings of the 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), Berlin, Germany, 28–31 October 2019; pp. 456–462.

49. Jiang, B.; Liu, Y.; Chan, W. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 3–7 September 2018; pp. 259–269.
50. Ashraf, I.; Ma, X.; Jiang, B.; Chan, W. GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities. *IEEE Access* **2020**, *8*, 99552–99564. [CrossRef]
51. He, J.; Balunović, M.; Ambroladze, N.; Tsankov, P.; Vechev, M. Learning to fuzz from symbolic execution with application to smart contracts. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 531–548.
52. Nguyen, T.D.; Pham, L.H.; Sun, J.; Lin, Y.; Minh, Q.T. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Valencia, Spain, 29 June–2 July 2020; pp. 778–788.
53. ethereum. ethereum/go-ethereum. 2021. Available online: <https://github.com/ethereum/go-ethereum> (accessed on 19 January 2022).
54. Rodler, M.; Li, W.; Karame, G.O.; Davi, L. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv* **2018**, arXiv:1812.05934.
55. Fattahi, S.M.; Makanju, A.; Milani Fard, A. SIMBA: An efficient simulator for blockchain applications. In Proceedings of the 2020 50th IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S), Valencia, Spain, 29 June–2 July 2020; pp. 51–52.
56. Murray, Y.; Anisi, D.A. Survey of formal verification methods for smart contracts on blockchain. In Proceedings of the 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS), Canary Islands, Spain, 24–26 June 2019; pp. 1–6.
57. Garfatta, I.; Klai, K.; Gaaloul, W.; Graiet, M. A Survey on Formal Verification for Solidity Smart Contracts. In Proceedings of the 2021 Australasian Computer Science Week Multiconference, Dunedin, New Zealand, 1–5 February 2021; pp. 1–10.
58. Mavridou, A.; Laszka, A. Designing secure ethereum smart contracts: A finite state machine based approach. In Proceedings of the International Conference on Financial Cryptography and Data Security, Nieuwpoort, Curaçao, 26 February–2 March 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 523–540.
59. Mavridou, A.; Laszka, A.; Stachtari, E.; Dubey, A. VeriSolid: Correct-by-design smart contracts for Ethereum. In Proceedings of the International Conference on Financial Cryptography and Data Security, Frigate Bay, St. Kitts and Nevis, 18–22 February 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 446–465.
60. Kalra, S.; Goel, S.; Dhawan, M.; Sharma, S. ZEUS: Analyzing Safety of Smart Contracts. In Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, CA, USA, 18–21 February 2018; The Internet Society: Reston, VA, USA, 2018.
61. Zhao, W. 30 Million Dollar: Ether Reported Stolen Due to Parity Wallet Breach. 2017. Available online: <https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach> (accessed on 19 January 2022).
62. Security, S.C. 2020. Available online: <https://swcregistry.io/> (accessed on 19 January 2022).
63. Corporation, M. 2021. Available online: <https://cwe.mitre.org/index.html> (accessed on 19 January 2022).
64. Banisadr, E. How \$800 k Evaporated from the PoWH Coin Ponzi Scheme Overnight. 2018. Available online: <https://medium.com/@ebanisadr/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530> (accessed on 19 January 2022).
65. Akentiev, A. Parity Multisig Hacked. Again—Chain. *Cloud Company Blog—Medium*; Cloud Company: Hong Kong, China, 2017.
66. Technologies, P. A Postmortem on the Parity Multi-Sig Library Self-Destruct. 2017. Available online: <https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/> (accessed on 19 January 2022).
67. King of the Ether Throne. 2016. Available online: <https://www.kingoftheether.com/postmortem.html> (accessed on 19 January 2022).
68. Qureshi, H. A hacker Stole \$31 M of Ether-How It Happened, and What It Means for Ethereum. 2017. Available online: <https://www.freecodecamp.org/news/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce/> (accessed on 19 January 2022).
69. CurrencyTycoon. Tricked by a Honeypot Contract or Beaten by Another Hacker. What Happened? 2018. Available online: [https://www.reddit.com/r/ethdev/comments/7x5rwr/tricked\\_by\\_a\\_honeypot\\_contract\\_or\\_beaten\\_by/](https://www.reddit.com/r/ethdev/comments/7x5rwr/tricked_by_a_honeypot_contract_or_beaten_by/) (accessed on 19 January 2022).
70. Bogatyy, I. Implementing Ethereum Trading Front-Runs on the Bancor Exchange in Python. 2017. Available online: <https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798> (accessed on 19 January 2022).
71. Reutov, A. Predicting Random Numbers in Ethereum Smart Contracts. 2018. Available online: <https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620> (accessed on 19 January 2022).
72. Bahrynovska, T. 2017. Available online: <https://applicature.com/blog/blockchain-technology/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes> (accessed on 19 January 2022).
73. Academy, K. Re-Entrancy Example. 2021. Available online: <https://solidity-by-example.org/hacks/re-entrancy/> (accessed on 19 January 2022).
74. Antonopoulos, A.M.; Wood, G. *Mastering Ethereum: Building Smart Contracts and Dapps*; O'reilly Media: Sebastopol, CA, USA, 2018.

75. OpenZeppelin. SafeMath Library. 2018. Available online: <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v1.12.0/contracts/math/SafeMath.sol> (accessed on 19 January 2022).
76. OpenZeppelin. 2020. Available online: <https://openzeppelin.com/> (accessed on 19 January 2022).
77. Academy, K. Delegatedcall Example. 2021. Available online: <https://solidity-by-example.org/hacks/delegatedcall/> (accessed on 19 January 2022).
78. Foundation, E. 2021. Available online: <https://docs.soliditylang.org/en/latest/contracts.html?highlight=library#libraries> (accessed on 19 January 2022).
79. ConsenSys. Ethereum Smart Contract Best Practices. 2021. Available online: <https://consensys.github.io/smart-contract-best-practices/> (accessed on 19 January 2022).
80. Tjaden Hess, P.C. How Can I Securely Generate a Random Number in my Smart Contract? 2016. Available online: <https://ethereum.stackexchange.com/questions/191/how-can-i-securely-generate-a-random-number-in-my-smart-contract> (accessed on 19 January 2022).
81. RANDAO. randao/randao. 2019. Available online: <https://github.com/randao/randao> (accessed on 19 January 2022).
82. Security Considerations. 2021. Available online: <https://docs.soliditylang.org/en/develop/security-considerations.html#tx-origin> (accessed on 19 January 2022).
83. Foundation, E. Introduction to Smart Contracts. Available online: <https://docs.soliditylang.org/en/v0.8.2/introduction-to-smart-contracts.html?highlight=storage#storage-memory-and-the-stack> (accessed on 19 January 2022).
84. Wiki, E. Mining Process. 2021. Available online: <https://eth.wiki/en/fundamentals/mining> (accessed on 19 January 2022).
85. He, D.; Deng, Z.; Zhang, Y.; Chan, S.; Cheng, Y.; Guizani, N. Smart Contract Vulnerability Analysis and Security Audit. *IEEE Netw.* **2020**, *34*, 276–282. [CrossRef]
86. Meiklejohn, S.; Pomarole, M.; Jordan, G.; Levchenko, K.; McCoy, D.; Voelker, G.M.; Savage, S. A fistful of bitcoins: Characterizing payments among men with no names. In Proceedings of the 2013 Conference on Internet Measurement Conference, Barcelona, Spain, 23–25 October 2013; pp. 127–140.
87. Ron, D.; Shamir, A. Quantitative analysis of the full bitcoin transaction graph. In Proceedings of the International Conference on Financial Cryptography and Data Security, Okinawa, Japan, 1–5 April 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 6–24.
88. Zhou, Y.; Kumar, D.; Bakshi, S.; Mason, J.; Miller, A.; Bailey, M. Erays: Reverse Engineering Ethereum’s Opaque Smart Contracts. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 1371–1385.
89. Community, E. What Is Oracles? 2021. Available online: <https://ethereum.org/en/developers/docs/oracles/#top> (accessed on 19 January 2022).
90. Community, E. Oracle Problems. 2021. Available online: <https://docs.ethhub.io/built-on-ethereum/oracles/what-are-oracles/> (accessed on 19 January 2022).
91. SmartContract Chainlink Ltd. 2021. Available online: <https://chain.link/> (accessed on 19 January 2022).
92. Parr, T. 2021. Available online: <https://www.antlr.org/> (accessed on 19 January 2022).
93. Aho, A.V.; Sethi, R.; Ullman, J.D. Compilers, principles, techniques. *Addison Wesley* **1986**, *7*, 9.
94. W3C. 2011. Available online: <https://www.w3.org/TR/xpath20/> (accessed on 19 January 2022).
95. Cavnar, W.B.; Trenkle, J.M. N-gram-based text categorization. In Proceedings of the SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval, Las Vegas, NV, USA, 11–13 April 1994.
96. Foundation, L. The LLVM Compiler Infrastructure Project. 2021. Available online: <https://llvm.org/> (accessed on 19 January 2022).
97. Authors, G.E. Geth. 2013. Available online: <https://geth.ethereum.org/> (accessed on 19 January 2022).
98. Zalewski, M. Technical “whitepaper” for afl-fuzz. 2017. Available online: <https://vyagers.com/2018/12/15/technical-whitepaper-for-afl-fuzz/> (accessed on 19 January 2022).
99. Authors, A. Aleth. 2021. Available online: <https://github.com/ethereum/aleth/> (accessed on 19 January 2022).
100. *Remix: An Online Smart Contract Development IDE*; Remix Inc.: Osaka, Japan, 2021.
101. Perez, D.; Livshits, B. Smart contract vulnerabilities: Does anyone care? *arXiv* **2019**, arXiv:1902.06710.
102. Chen, J.; Xia, X.; Lo, D.; Grundy, J.; Luo, X.; Chen, T. Defining smart contract defects on ethereum. *IEEE Trans. Softw. Eng.* **2020**, *48*, 327–345. [CrossRef]
103. Proposals, E.I. Ethereum Improvement Proposals. 2021. Available online: <https://eips.ethereum.org/> (accessed on 19 January 2022).