A Survey on Vulnerabilities of Ethereum Smart Contracts

Zulfiqar Ali Khan and Akbar Siami Namin

Department of Computer Science
Texas Tech University
zulfi.khan, akbar.namin@ttu.edu

Abstract

Smart contract (SC) is an extension of BlockChain technology. Ethereum BlockChain was the first to incorporate SC and thus started a new era of crypto-currencies and electronic transactions. Solidity helps to program the SCs. Still, soon after Solidity’s emergence in 2014, Solidity-based SCs suffered many attacks that deprived the SC account holders of their precious funds. The main reason for these attacks was the presence of vulnerabilities in SC. This paper discusses SC vulnerabilities and classifies them according to the domain knowledge of the faulty operations. This classification is a source of reminding developers and software engineers that for SC’s safety, each SC requires proper testing with effective tools to catch those classes’ vulnerabilities.

Index Terms

Smart Contract, Ethereum, EVM, vulnerabilities, Solidity, tools

I. INTRODUCTION

BlockChain is the most significant development to promote crypto-currencies. There are variations of BlockChain. Ethereum BlockChain, also known as Ethereum Virtual Machine (EVM) allows various unknown individuals to join hands and work together under a digital agreement known as a SC. Contracts require rules, but in this case, a programming language called Solidity embeds the rules within the SC itself. SC does not contain any “main” method, so it’s not self-executable.

SC is deterministic. This constraint generates the same output when any node of the Ethereum network executes the SC. Nodes can be users, or nodes can be miners responsible for validating SCs by solving a mathematical puzzle. The mathematical puzzle should generate the same result on all nodes, and this validation process follows the consensus protocol.

Once the mining process completes, its owner uploads the SC on the BlockChain. If the contract does not fit according to Ethereum rules, miners discard it. This process might follow the re-submission of SC. Thus the consensus protocol becomes a method for developing trust among the parties.

The trust is that there exists no error or fraud. But once the owner uploads the SC, it becomes immutable. Thus an unsafe SC can shatter the trust. Hackers can misuse it, causing considerable losses to the SC account holders. Therefore it is necessary to identify the vulnerabilities of SC before one uploads the SC on the BlockChain.

This survey research paper focuses on 20 vulnerability patterns. We have also provided the Solidity code corresponding to each vulnerability. We have used the context of SC’s faulty operation for the classification of vulnerabilities. For brevity reasons, we have skipped the mitigation techniques.

II. MOTIVATION

Previous research work has created several taxonomies related to vulnerabilities of SCs. The typical approach is to categorize the vulnerabilities based upon EVM, BlockChain, and Solidity associated issues as discussed in [1]. The above three are broader classes and give less information to the readers about the code’s internal drawback.

Similarly, the survey in [2] classifies the SC vulnerabilities in the context of EVM and Solidity. Other significant survey grouped the vulnerabilities based upon layering [3] like application layer, data layer, and consensus layer. Furthermore, the work in [4] used NIST Bugs Framework for the classification. However, SC introduced new kinds of vulnerabilities [3] not common in traditional programming paradigms.

We are the first to classify SC’s security issues according to the vulnerable operation’s domain knowledge. A good understanding of vulnerabilities requires coding examples along with description. Work in [2], [3] also provided coding examples, but we address a different set of vulnerabilities. It is worth mentioning that the most relevant survey on the SC attacks with vulnerable SC code is available in [5], published in 2017. Since then, the Solidity programming language has undergone major changes. In this survey paper, we replicate vulnerabilities in Solidity code using the “solc” compiler version 5.0. The key contributions of this survey paper are:

– A classification of the SC vulnerabilities according to the domain knowledge.
– A description of SC vulnerabilities through updated sample code.
– A discussion of twenty SC vulnerabilities along with twenty-four SC vulnerability detection tools.
– A discussion on the regeneration of deprecated Var vulnerability (section IV-D2).
The remainder of this paper is organized as follows: section III presents our classification of SC’s vulnerabilities. Section IV presents the description of SC vulnerabilities with respect to domain knowledge along with some sample code. Section V lists the testing tools developed for SCs. Section VI concludes the paper and highlights the future research directions.

### III. A Classification of SC’s Vulnerabilities Based on Domain Knowledge

SCs are susceptible to coding threats. One research classifies the vulnerabilities into security, functional, operational, and developmental categories [6]. These categories do not necessarily reflect any knowledge specific to SC.

Our research classifies the vulnerabilities using the domain knowledge of operations performed by vulnerable SC. For instance, SCs communicate with each other by invoking methods. This inter-SC communication operation can pave the way for several coding irregularities like reentrancy, denial of service, mishandled exception, and so on. In the same way, other SC specific operational knowledge helped us to create vulnerability categories like Contractual, Arithmetic, Gas Related, Transactional, Randomization and Deprecated vulnerabilities. The domain knowledge-based classification provides (i) more knowledge about the cause of vulnerability (as compared to previous taxonomies discussed above) and (ii) we can drill down to the specific instructions (in some cases). For instance, in case of inter-contractual vulnerabilities, we can infer that call is behind this vulnerability. Due to space limits, we have excluded randomization and deprecated vulnerabilities from the discussion.

Figure 1 depicts our classifications of SC’s vulnerabilities. As shown in Figure 1, the SC’s vulnerabilities are grouped into 1) inter-contractual, 2) contractual, 3) integer bugs, 4) gas related, 5) transnational, 6) deprecated, and 7) randomization vulnerabilities.

### IV. Descriptions of SC Vulnerabilities

#### A. Inter-Contractual Vulnerabilities

Listing 1 shows an example of inter-contractual communication between ModifiedBank and ModifiedMalicious SCs using call modified from [7] in the context of a bank and an attacker. call can also generate reentrancy attack. Solidity provides send, transfer, and call functions for contract-wide Ether transfers along with an external payable fallback function (FF) at the receiving SC. FF is an anonymous function, which retrieves the transferred Ether in the global state variable msg.value. Transferring Ether incurs gas charges. send and transfer can compensate for 2300 [1] amount of gas which is sufficient only for executing FF not linked to state changes. call can transfer entire gas, which can initiate reentrancy attack, as discussed below:

```
pragma solidity ^0.5.1;
contract ModifiedMalicious{
    ModifiedBank mb;
    constructor(address payable addressOfBank, uint amount) public{
```

1This paper lists the EVM opcodes in capital letters and references the Solidity functions and contents of Listings in this font.
Fig. 2. The reentrancy cycle, (Solidity-like) code modified from [7], [12].

```
5  mb=ModifiedBank(addressOfBank);
6  mb.withdraw(amount);
7  }
8  function () external payable{ mb.withdraw(msg.value);}
9  }
10 contract ModifiedBank{
11  mapping(address=>uint)bal;
12  //more
13  function withdraw(uint _amount) public returns (bytes memory message) {
14  if(bal[msg.sender]>= _amount){
15  (bool success, bytes memory returnsMessage) = msg.sender.call.value(_amount)();
16  bal[msg.sender]-=_amount;
17  /*more*/}
18  function () external payable{}}
```

Listing 1. Example Solidity Code of Reentrancy, modified from [7].

1) Reentrancy: Reentrancy is a devastating attempt to deprive the investors of their precious cash. Even though the reentrancy attack (or the DAO attack [8]) did not knock out the newly born SC technology, but created a stir in the Ethereum community. Reentrancy problem soon became an exciting topic of discourse among the Blockchain researchers because of two reasons: its inherent characteristics (discussed below) and the enormity of Ether flown out from SC as a result of it.

Reentrancy problem occurs when an attacker reenters the SC repeatedly. This reentrance sparks of multiple issues like fueling out the entire gas of the victim, hijacking of the victim’s SC , execution of two functions at the same time and undue transfer of funds from victim’s SC to the attacker’s account. Developmental issues coupled with the built-in implicit invoking nature of FF provide a stimulus for this attack.

There are several variants of this type of attacks like same or cross-function reentrancy ([9]), “Single-entrancy” ([10]), and the state-of-the-art reentrancy attacks discussed in [11]. Reentrancy is a repeated operation and the statements execute in a cycle as shown in Figure 2. As discussed above, Listing 1 tries to simulate the banking application. However, the banking contract provides only the withdraw(...) method with a flaw. The attacker ModifiedMalicious exploits this flaw to drain the Ether from the bank into its own account.

Listing 1 demonstrates how the ModifiedMalicious contract exploits the flaw in line#16. Firstly, the attacker’s SC ModifiedMalicious calls the ModifiedBank SC’s withdraw method in line#6 to retrieve balance amount from her SC account. As the SC ModifiedBank executes line#15 (using call) for transferring, the action results in the voluntary invoking of the costly FF of the attacker in line#8. It is costly because the FF in line#8 is not empty and contains the code to invoke the withdraw method of ModifiedBank in line#8. This process repeats until the bank or the attacker reaches out of gas state. However, in the meantime attacker may drain handsome amount from the bank because the debit process never deducts her balance, i.e., line#16 never executes.

2) Denial of Service (Unexpected throw): Denial of Service occurs due to various reasons. External functions may contain broken linkage due to the use of throw (deprecated). Listing 2 (excerpted from [5]) shows the contracts (i.e., MKotET1, MKotET1_1). Both are related to a game to acquire the throne of King of Ether. The throne’s price is the money required by current king to leave the throne along with some processing fee to the SC’s owner. However both the contracts suffer from Denial of Service vulnerability. MKotET1 (starting from line#1) exhibits Denial of Service threat using transfer in line#7 and MKotET1_1 (starting from line#12) displays Denial of Service threat using call in line#17. MKotET1 and MKotET1_1 use transfer and call respectively to deliver Ether to SC MMallory (starting from line#22), which contains a risky FF,
in line#23, which reverts in line#24, throwing an exception unconditionally. Both transfer and call fail to deliver Ether to ‘MMallory’ due to the revert in line#24.

Listing 2. KingOfEtherThrone threat variants: Transfer, Call

3) Mishandled Exception: Mishandled Exception [13] is a frequently occurring threat [14] and appears by other names in the surveyed literature such as “Unchecked send”, “Unchecked External Call”, and “Exception Disorders.” Exceptions are run-time errors. One of the well-known exceptions in the Ethereum network is the out-of-gas exception. However, when a SC invokes an untrusted external function, a programmer must take extra care. For instance, sending Ether, implicitly invokes the FF of another SC. FF may fail and the reason might not be the out-of-gas exception [15].

In Solidity, we can use transfer, send, and call for sending Ether to another SC. If an exception occurs in the callee, transfer propagates the exception in the caller’s SC, which is safe. APIs like call, send, and delegatecall return false and the execution continues [16]. Thus, if the programmer skips the checking of the false returned value, execution would continue resulting in an inconsistent state [17]. Surveyed literature argues that the owner deliberately does not throw an exception if the send operation fails. This attitude may result in exceeding the call-depth stack (i.e. CDS, which is a deprecated vulnerability) [5], [18]. MMallory contract in Listing 3 undo the transfer of Ether to itself. The payable directive facilitates the importing of Ether, as discussed in section IV-B under frozen Ether vulnerability. However, anybody sending Ether to MMallory will suffer from exception due to revert in line#2.

Listing 3. ‘revert’ in line#2, causes mishandled exception [5] to the caller.

4) Gasless send [13]: send is associated with a fixed gas stipend of 2300 [12], which is enough to execute an empty FF. If the FF modifies the state of the SC, then the required gas can increase beyond 2300. This costly FF results in an out-of-gas exception [5]. Costly FF can be due to the developer’s mistake instead of a malicious activity [19]. In other words, if the “gas used” is high, an exception occurs, and the malicious miner may keep the untransferred amount.

Listing 4. Receiver SC’s FF is costly: alters ‘TotalBal’, line#7. [20].

But in some cases, the malicious SC owner may keep the non-transferred amount and miner only gets his fee. call mitigates this threat but transfers all gas which causes reentrancy. Listing 4 shows two SCs, Sender line#1 and Receiver line#4 modified from [5]. Sender transfers funds to Receiver using send in line#3 but Receiver retrieves the Ether using a costly FF in line#6. Transfer increments the state variable TotalBal in line#7. But transfer in line#3 becomes a threat as send uses 2300 amount of gas, which is not enough to execute a costly FF.
5) call Transfers All Gas: At the bytecode level, Solidity’s send and transfer translate into CALL (EVM bytecode). CALL executes the FF of the calling SC and can fail due to insufficient gas. It consists of four stack arguments: 1) the amount of gas required for the transaction, recipient’s address, 2) the amount of Ether to be transferred, 3) size, and 4) the location of input data along with the size and location of return data. call can provide the stimulus for reentrancy attack by forwarding all the gas. However, implicit call statements are also possible. Line #7 uses call implicitly, which is vulnerable; the first parameter is the Ether value, while the second parameter is the argument of f().

Listing 5. test2: implicit CALL sends amount to f(), in line#7.

B. Contractual Vulnerabilities

Contractural bugs impact the SC itself. Both the attacker and owner exploit them for causing harm to the SC users. The owner can design the SC to prevent leakage (or transfer) of any Ether, thus turning the SC into a black hole. However, the worst happens when an attacker uses the unprotected selfdestruct command to destroy the SC. The attacker enjoys the balance of the account if the attacker changes the ownership of SC before destroying it.

1) Frozen Ether: Also known as “Locked Money” or “be no black hole”. The frozen Ether threat deprives the SC account holders of Ether worth millions of dollars, as in the case of parity wallet SC. One atypical impact of the above exploits resulted after the accidental killing of the library SC, which provided an external route to Ether in parity Wallet SC (and to other multisigWallet-like SCs).

Apart from the accidental execution of suicide command, which the hobbyist confessed of doing in his issue# 6995 on Github, there could be coding loopholes in the SC, which can prevent exporting of Ether from the SC. Programmatically, exporting Ether weakness applies to SCs, which lack statements like call, send, or transfer, which move the funds outside the SC, along with the presence of payable directive in the SC, which on the other hand facilitates importing of Ether. To summarize, “frozen Ether” vulnerability occurs when:

1) SC permits inbound Ether traffic but shuts the outbound Ether traffic. Reference labels such SCs as greedy. FF handles inbound Ether traffic but call, send and transfer handle outbound Ether traffic. Listing 6 uses a SC modified from, which shows the freezing Ether vulnerability because the SC contains a method having payable directive in line#2 but does not contain program paths leading to CALL, DELEGATECALL, or SELFDESTRUCT opcodes:

Listing 6. payable directive creates an Ether receiving FF in line#2.

2) Wallet SC relies on another SC or library. The library SC provides functions to support the Wallet, for instance, the library can provide function for transferring Ether. But if the library SC eventually kills herself by executing selfdestruct command or some other SC (or an attacker) accidentally (or deliberately) kills the library SC, then it would close the doors of Ether extraction from the wallet SC. Listing 7 modified from, shows the use of delegatecall in line#4 to load the code from the address 0xNewLibrary, line#2, containing the library’s withdraw() method.

Listing 7. withdrawM() uses library through hardcoded address(line#2) and delegatecall (line#4) using that address.

2) Self-Destructible: selfdestruct (previously known as suicide) allows a SC to destroy itself by releasing the Ether of account holders. SC uses this alternative in emergencies. Research conducted by terms a SC suicidal (i.e. vulnerable for SELFDESTRUCT opcode) if the SC does not correctly guard the selfdestruct command. Attacker requires two things to kill a SC: (1) reaching into the conditional statement enclosing the selfdestruct statement, and (2) attaining the ownership of SC. selfdestruct causes unresponsiveness of the SC resulting in “Denial of Service.” selfdestruct deletes the SC’s code permanently. All the SC’s funds would transfer to the account associated with SC. This transfer will not trigger
FF [28]. A beneficiary can be an existing account or the account may not exist. In the latter case, the destruction process creates the account and charges fees for it [21]. Contract MDiscontinue in Listing 8 provides a TerminateMe function to destroy the contract with unprotected selfdestruct (i.e., without any if block) in line#6, modified from [30]:

```
contract MDiscontinue{
    address payable owner;
    constructor () public {
        owner = msg.sender;
    } function TerminateMe () public {
        selfdestruct (owner);}//suicidal
```

Listing 8. selfdestruct, line#6, without any guard, suicidal SC [30].

3) Stealing Ether: Stealing Ether vulnerability (or Unsecured Balance) surfaces when the SC initializes the owner field indirectly as in line#4 of Listing 9 in a function other than the constructor. The use of a function for initialization of owner address (as in line#3) can lead to a problematic situation. This situation is similar to parity SC, which doomed the multi-signature SC of 30m dollars [31].

```
contract CompWallet{
    address payable owner;//state variable
    function initComWallet (address payable _owner) public{
        owner = _owner;}//any user can change owner
    function withdraw (uint _amount) public {
        if (msg.sender == owner){
            if (!owner.send(_amount)){}
        } else {}}
```

Listing 9. owner initialized in line#4, outside constructor [27].

C. Arithmetic Bugs

This problem occurs as a result of a mathematical operation. Most significant bug is the integer overflow/underflow, which is a common problem in programming languages. Increments beyond the maximum or decrements below the lowest value (i.e. wrap-around) may generate wrong results. Thus, developers must perform manual checking (i.e., employ a human expert to check the code); otherwise, code may create a wrap-around error. One solution is to use SafeMath library. The latest research recommends using the “solc-verifier” tool.

1) Integer Overflow/Underflow: Ethereum has nothing to do with data types leaving the compiler responsible for catching integer overflows and under-flows [17], [32]. Solidity is rich in integer data types with flexible sizes (uint8, uint16, uint24,..uint256, int8, int16, int24,..int256) but does not support floating-point math [6]. Thus, the diversity of integer data types provides no benefit to SC. Work in [17], [33] provides an example code generating integer bugs. Figure 3 shows an underflow SC and its output on Remix (0 changes to 255 on decrements).

```
pragma solidity ^0.5.1;

contract MUFTest1 {
    uint8 testVal = 0;
    function testFunc() public returns (uint8) {
        testVal--;
        return testVal;}
    //Debugging on Remix IDE
    Decoded output {
        "0": "uint8: 255"
    } 0:0x000000000000000000000000000000000000000000000000000000000000000ff
```

Fig. 3. Underflow SC, MUFTest1, modified from [34], and the result of debugging it on REMIX IDE.

SafeMath [6], [35] provides several functions to replace the ordinary arithmetic operations in SCs. Listing 10 lines#2-4, show the logic of SafeMath library’s method sub(·). Figure 3 SC, MUFTest1, is modified from [34]. We replaced the line#5 in Figure 3 by sub(·) method of SafeMath library as shown in Listing 10 line#9, of SC MUFTest2.
2) Unchecked Maths: Unchecked math means that a SC is not using strategies to protect mathematical statements from overflows/underflows. A good practice is to protect the code using assertions and SafeMath library as shown again in Listing 10 line#9.

```solidity
library SafeMath {
    function sub(uint8 x, uint8 y) internal pure returns (uint8) {
        assert(y <= x);
        return x - y; }
    contract MUFTest2 {
        using SafeMath for uint8;
        uint8 testVal = 0;
        function Utest() public returns (uint8)
            testVal = testVal.sub(1);
        //instead of : testVal--; (in line#5, Fig. 3)
        return testVal; }
}
```

Listing 10. Use of sub(...) function, line#9, to avoid underflow [34].

D. Gas Related Issues

The gas serves two essential purposes for the EVM network. Firstly, gas serves as compensation to the miners’ efforts for recording transactions. Secondly, the gas acts as a fuel for running a transaction and thus prevents long transactions from hijacking the EVM scheduling scheme. Logically, it means that if the user does not pay enough gas fee as required for the transaction, the transaction will fail by generating an “out of gas” exception. Surprisingly, an integer overflow can also cause “out of gas.” Other examples are Denial of Service, and Wallet Griefing, as discussed below:

1) Denial of Service (Costly Loops Causing Out of Gas Exception): EVM protects programs from Denial of Service attacks by forced termination. EVM allocates gas at the start of execution, and each execution step results in some deduction. If the remaining amount after deduction is less than the amount required for execution, EVM [15] terminates the SC’s execution, causing partial or full rollback [36]. The termination can occur even without the influence of an attacker [37].

An attacker can manipulate the arr, line#4, in Listing 11 by adding additional addresses. Thus, increasing the execution cost and transfer to manipulated addresses. In the worst case, the entire gas may be exhausted, resulting in full revert. Denial of Service can occur due to the presence of revert in external function, as in Listing 2 line#24.

```solidity
contract testLL{
    uint constant LARGEGAS = 100000;
    address payable addrArr;
    function LongList(uint256 memory nextV, uint[] memory arr, address payable _addr) public {
        uint256 i= nextV;
        addrArr = _addr;
        for (;i < arr.length && gasleft() > LARGEGAS; i++) {
            addrArr.send(addr(i));
        }
        nextV = i;
    }
}
```

Listing 11. Unbounded mass operation: traversing an un sized array [15].

Another example is the Denial of Service due to a costly for or a while loop. This may cause depletion of gas in each iteration and finally resulting in Denial of Service. Work presented by [15] renames this vulnerability, as “Unbounded mass operations due to an un sized array variable arr in line#7. Listing 11 and [15] have proposed resumable loops. Function LongList(..), line#4, in Listing 11 is excerpted from [15]. In the case of revert, due to “out of gas” threat, nextV, in line#9, points to the arr index, from where to resume.

2) Integer Overflow (Causing Out of Gas Exception): Authors of SmartCheck [6] and MadMax [15] discuss about the integer overflow issue in a loop in the context of Var (deprecated, Solidity used Var for Type Inference [6], [9]). However, an integer overflow can occur even when Var does not determine the type of loop index (i.e., uint8) variable. By this, we mean that integer overflow can occur if the type of loop index variable is a short integer at run-time. Listing 12 shows an overflow without using Var, as in line#4 in Listing 12 when the loop counter variable becomes greater than or equal to 255.

```solidity
contract Overflow {
    int [300] emp;
    function testOF() public returns (bool) {
        for (uint8 i = 0; i < emp.length; ++i) { }}
}
```

Listing 12. Short integer overflows even without Var [38] in line#4.

3) Wallet Griefing Causing Out of Gas Exception: This vulnerability occurs if a SC uses a loop to send Ether to multiple SCs. Thus, if one receiver fails, then the entire transaction fails. The receiver can fail due to a bad FF (line#2, Listing 3). If the sending SC uses a throw (i.e., revert) to handle the failure of send then it can exacerbate the situation because throw consumes the entire gas [14], [39] and locks the sender’s SC [15], [40]. Repeated attempts may also fail due to “out of gas” situation, as in line#6, Listing 13 modified from [15]. However, the latest version in can achieve the same effect by using require and transfer instead of send and revert.
for ( uint i = 0; i < employee.length ; i++) {
    if ( employee [i].paid < min_salary ) {
        // sentinel for making a payment.
        // code may lock the SC
        // due to {vtt revert} consuming all gas.
        if (!( employee [i].addr.send( employee [i].bonusAmount ))) revert();
        employee [i] = newEmployee ;
    }
}

Listing 13. Wallet Griefing using revert, send fails in line#6. [15].

E. Transactional Irregularities

EVM transactions become a source of greed for the miner, which validates them. Thus some miners can influence the transactions resulting in vulnerabilities like Transaction Ordering Dependence and Time Stamp Dependence.

1) Transaction Ordering Dependence (TOD): TOD is also known as “front running race condition” [16]. This attack, “selfish mining attack”, occurs due to the mishandling of transaction queue by miners. The owner/user incentivizes the miner to change the order of the transaction [18].

One example of TOD is the case of marketplace SC as shown in Listing 14 modified from [18] in which a miner may not honor a leading buyer’s request at the cost of some other transaction. The buyer sends the transaction to buy at cost 100, line#7, Listing 14. At the same time, owner sends the transaction with a high gas fee to increase the price, line#4, Listing 14. The owner’s transaction executes first due to the higher gas price incentive. The buyer’s transaction completes next but the buyer pays more.

Listing 14. Miner alters lines#5-8, causing TOD: marketplace [18].

Contrary to the general notion about a miner in connection with TOD, the recent research conducted by [17] argues that it is hard to exploit TOD threat because the attacker should be a miner, and there are less financial gains. Apart from this, there are some other programming problems. State variables often have a dependency on the function, which changes their values. Thus, coding such a function when multiple users are invoking that function is a concurrent programming issue.

The concurrency in SCs requires some mechanism like semaphore to control access to the state variable. Reference [9] argues that Solidity does not support concurrency. Thus, manipulation by a miner is a limitation of BlockChain rather than a bug. However, EVM must provide some solution for miner’s problems, as the miners also contribute to immense power consumption. One solution proposed by [41] is to delegate miners’ role to a SC. Another name for this problem is the unpredictable state problem. This is because multiple invocations of a dependent function make it difficult to predict what the state and the values stored within a SC will be when a user executes the function.

2) TimeStamp Dependence [13]: Each block within the BlockChain contains three pieces of information: 1) timestamp, 2) cryptographic hash, and 3) the transaction data.

− Timestamp represents the time when the miner verifies all the transactions within the block after computing the proof-of-work puzzle. A miner can manipulate the block timestamp but has to complete the validation within 900s [18]; otherwise other miners would reject the block.

− Cryptographic hashesare deterministic functions. Miners exploit the deterministic hash values to verify the integrity of the block’s data. Cryptographic hash chains the current block with the previous block as there is a dependency between hash values of the two said blocks.

− Transaction data can vary based upon transactions. Still, for the most straightforward transactions between two SC accounts, the transaction data would be the sender’s and receiver’s SC account addresses and the amount of Ether sender transfers to the receiver.

Despite the doubtful accuracy, SCs use the timestamp for random number generation. Due to the miner’s involvement in setting the timestamp’s value, the timestamp becomes a so-called deterministic random value. Hence, the usage of the timestamp as a random value in lottery implementation is vulnerable [21]. Programmatically, block.timeStamp retrieves the timestamp associated with a block. However, one can just use now (alias for block.timestamp) also to retrieve the timestamp, as in Listing 15 lines#3-4. Solidity uses now for simplicity and bytecode implementation. But Solidity does not discriminate between now and block.timeStamp and hence both are vulnerable.
contract TSD {
  function Mpay () public {
    uint tTime = now;
    if (tTime > (now + 2)) { if(msg.sender.send(200)) } else {} }
}

Listing 15. Miner’s misuse of now (lines #3-4) causes TSD.

3) tx.origin: tx.origin is a transaction state variable which indicates the originator of the transaction. Other transaction state variables also exist like tx.GasPrice. But tx.GasPrice has a fixed value so the adversary cannot change it [9]. On the other hand, tx.origin can vary. This variation can lead to attacks because we cannot use tx.origin to ratify the contract’s owner, as shown in Listing [16].

Lines #1-9 in Listing [16] show the victim’s SC (i.e., TxUserWallet) and lines #10-11 show the interface (i.e., TxUserWallet). Note that the names of the victim’s SC and the interface are the same but they are in different files.

contract TxUserWallet {
  address owner;
  constructor () public {
    owner = msg.sender;
  }
  function sendTo(address dest, uint256 amount) payable public returns (bytes memory theMessage) {
    require(tx.origin == owner);
    (bool success, bytes memory returnMessage) = dest.call.value(amount)();
    require(success);
    return returnMessage; } /* more */
}

Listing 16. Example of tx.origin [12], Victim’s SC (i.e., TxUserWallet).

The rest of the code from lines #12-17 show the attacker’s SC, TxAttackWallet. The surveyed literature recommends the replacement of tx.origin with msg.sender particularly for authenticating the sender of a message [14]. tx.origin represents the address of the first account in the call chain (i.e., the list of calls related to the currently executing transaction), whereas msg.sender is the original caller [9].

V. TOOLS FOR TESTING SMART CONTRACTS

we provide a brief description of SC tools developed for detecting above mentioned vulnerabilities. We grouped the testing tools into dynamic and static-based tools.

A. Testing Tools based on Static Analysis of SCs

There are a good number of static-analysis for testing SCs:

– **Zeus** [9]. Zeus is a tool for formal verification of SCs using abstract interpretation and symbolic model checking. Zeus works directly on the high-level of SC code. Zeus detects threats like reentrancy (section [IV-A1]), Unchecked and Failed send (section [IV-A3]), Integer Overflows (section [IV-C1]), and Timestamp dependency (section [IV-E2]).

– **VeriSolid** [43]. VeriSolid is a SC development tool. VeriSolid uses a transition system model to generate Solidity-based formally verified SCs. Automatic code generation is an important achievement in the context of formal verification tools. VeriSolid prevents reentrancy (section [IV-A1]) by design and uses liveness property to prevent Denial of Service (section [IV-A2], [IV-D1]).

– **Vandal** [14]. Vandal is a static analysis tool. Vandal performs security analysis of EVM bytecode using a logic language, called Souffle, to transform the analysis into C++. Vandal’s static analysis library functions detect threats like “Unchecked Send” (section [IV-A3]), Reentrancy (section [IV-A1]), UnSecured Balance (section [IV-B3]), Destroyable SC (section [IV-B2]).

– **Teether** [22]. Teether focuses on the same idea of critical paths as in [24]. Teether constructs the CFG of the SC using the EVM bytecode, which helps to detect critical paths. The authors discussed the peculiar problem of backward traversal associated with JMP because of JMP’s similarity with x86’s return statement.

– **SmartScopy** [44]. SmartScopy is an attack synthesizer. SmartScopy performs summary-based symbolic evaluation, which reduces the program size for symbolic analysis (SA) required for automatic generation of adversarial SC. The adversarial SC confirms the presence of vulnerability in the victim SC, detected by manual analysis. SmartScopy detects threats like reentrancy (section [IV-A1]), timestamp dependence (section [IV-E2]), and Gasless send (section [IV-A4]).
- **sCompile** [24]. sCompile exploits the notion of “critical paths” i.e., in place of identifying a program to be vulnerable, recommends using more sophisticated techniques like taint analysis (TA). The authors identified several coding threats like reentrancy (section IV-A1), Mishandled Exception (section IV-A3), call transfers all gas, and so on. The SmartCheck [6] related research provides a comprehensive list of threats based upon exploits related to security, functional, operational, and developmental issues.

- **Securify** [27]. Securify is a static analysis tool, focusing on patterns (compliance or violation). The tool extracts the domain knowledge from patterns related to a security property. Securify detects threats like Stealing (section IV-B3) and frozen Ether (section IV-B1), Reentrancy (section IV-A1), Mishandled Exception (section IV-A3) and Transaction Ordering Dependence (section IV-E1). Related violation properties with regarding threats are restricted write, Ether liquidity, no writes after calls, handled exception, restricted transfer and TOD [27].

- **Oyente** [18]. Oyente performs static analysis of SCs. [18] recommend the extension of Oyente and is a reality when one reads the details of the tools mentioned in [45]. Oyente detects threats like TOD (section IV-E1), TimeStamp Dependence (section IV-E1), and Mishandled Exception (section IV-A3).

- **OSIRIS** [32]. OSIRIS employs a strategy based upon taint analysis and symbolic execution and consists of an integer detection module. Symbolic execution module constructs a control flow graph (CFG) from the bytecode. The CFG processes different paths of the SC using symbolic values, as in Maian [25]. Osiris [32] detects arithmetic bugs (overflow/underflow (section IV-C1) and division by 0), truncation bugs (converting from a larger to smaller data size, e.g., 64-bit data to 16-bit data), and signedness bugs (converting a signed integer to an unsigned integer of the same width and vice versa).

- **MadMax** [15]. MadMax focuses on automatic detection of gas-focused threats. Like Vandal [14], MadMax performs the de-compilation of EVM bytecode and similarly uses a logic-based approach to produce a high-level representation of the program model. MadMax defines strategies for surviving out of gas conditions in the context of resumable loops (section IV-D1), loops bounded by induction variable, and dynamically bounded loops.

- **KFrameWork-EVM-semantics (KEVM)** [36]. KEVM is a semantic analysis tool based upon SE. KEVM’s development framework integrates a semantic debugger and a program verifier. Tool encapsulates a gas analyzer that computes gas bounds during execution and can help in detecting “Denial of Service” (section IV-A2) threats.

- **Interactive Theorem Provers (ITP)** [46]. Interactive Theorem Prover combines the idea of theorem proving with testing. Initially, this tool presents the desired behavior of EVM in LEM [47]. Authors used both the community-based test suits and interactive theorem provers like Isabelle/HOL to test their EVM definitions. The authors divided the formalization into deterministic and non-deterministic formalization. The basic assumption for non-deterministic formalization was to segregate the environment from the system. This helped to reason about the adversarial attack and to model the reentrancy attack (section IV-A1).

- **Gasper** [48]. Gasper uses Oyente Engine to generate the CFG and identifies the code for optimization. [48] points out seven gas costly-patterns. Due to shortage of space, we have not discussed Gasper patterns in this paper.

- **FSolidM** [49]. FSolidM allows development of secure SCs using Finite State Machines (FSMs) [16]. FSolidM uses a set of plugins and design patterns that developers can add to the SC for implementing locking, maintaining transaction counter, and enforcing timed transitions to safeguard against reentrancy (section IV-A1), transaction ordering dependence (section IV-E1), and time constraint, respectively.

- **FVF*** [7]. The research work in [7] describes the verification of Ethereum SCs using $F^*$ (FVF* stands for formal verification using $F^*$). The work in [7] added an effect system in $F^*$ which helps in the detection of “unchecked send” (section IV-A3) and destructive patterns like reentrancy (section IV-A1).

- **EtherTrust** [50]. Authors used the tool to prove the reachability property for SC’s bytecode. EtherTrust ensures two things about the SC: (i) FFs should not result in DAO type of attack (section IV-A1) (ii) data is not vulnerable to miner’s manipulation.

- **DappGaurd** [17]. DappGaurd detects the diverse type of threats. However, instead of relying on bytecode or Solidity code, DappGaurd focuses on Transaction Receipts, which analyze live SCs, but the authors do not provide the source for retrieving TRs. DappGaurd’s prototype version detects several threats, and for this purpose, DappGaurd incorporates the Oyente engine.

- **sCompile** [24]. sCompile identifies critical paths (i.e., money related inter-contractual paths involving call) in the program. Developed in C++, sCompile uses Z3 SMT Solver for SE. sCompile detects threats like reentrancy (section IV-A1), be no black hole (section IV-B1) and unguarded selfdestruct (section IV-B2).

### B. Testing Tools based on Dynamic Analysis of SCs

- **Vultron** [51]. Vultron is still in infancy stages and the authors have tested the prototype using truffle suite. Vultron is a test oracle, which stores bookkeeping information in variables. Vultron compares these variables with account balances related to SCs to determine the inconsistencies. Vultron can identify threats like reentrancy (section IV-A1), exception disorder (section IV-A3), integer overflow/underflow (section IV-C1), and “gasless send” (section IV-A4).
- **Sereum** [11]. Sereum modifies the “goethereum” client “geth” and adds an attack detector and taint engine. Sereum works at the bytecode level and the binary level does not keep type information. This fact makes it challenging to infer about the sensitivity of data [30]. Sereum related research reports a reentrancy attack (section IV-A1).

- **Reguard** [52]. Reguard is a dynamic analysis tool and incorporates a fuzzing based analyzer. Reguard focuses on automatic detection of common threats in SCs like reentrancy bugs (section IV-A1). Reguard transforms the code into an intermediate representation (IR) (Abstract Syntax Tree). Finally, Reguard executes the SC (with transactions as input) and forwards the dump of relevant operations of run-time analysis to the core detector to detect reentrancy bugs.

- **Maian** [25]. Maian is a static analysis tool but also performs dynamic analysis of SC. For offline inspection, Maian’s input is the EVM bytecode and the SC’s initial state retrieved from the Blockchain. SA generates actual values for the transaction given SC’s bytecode and analysis specification i.e., vulnerability category to search like Suicidal (same as selfdestruct, IV-B2) or Greedy (same as Frozen Ether, section IV-B1) as input. Maian then uses the actual values in the validation step.

- **EasyFlow** [35]. EasyFlow is a specialized tool focusing on integer overflow. EasyFlow uses taint analysis for overflow detection. Detection algorithm analyzes transaction instructions and mathematical instructions (at bytecode level) like EXP, ADDMOD, and MULMOD and even the instructions protected by SafeMath library.

- **ContractFuzzer** [72]. Fuzzing is a technique which can perform both static and dynamic analysis independently or at the same time. [53] discusses an example of fuzzing with SC by mounting the Truffle project to a docker image. ContractFuzzer detects threats like “Gasless send” (section IV-A4), “Exception disorder” (section IV-A3), Reentrancy (section IV-A1), TSD (section IV-E2), and Freezing Ether (section IV-B1).

### VI. Conclusion and Research Directions

We have provided classification of SC vulnerabilities but this can be further enhanced. In fact, security of SC is vital for strengthening the concept of Blockchain. Consistent efforts from academia have provided great solutions and this should continue. Our future research would be related to identifying randomization vulnerabilities. Following are the suggestion to fill the gaps in previous research and to advance the current research:

- For EVM researcher: a) there is a need for run-time environment within the Ethereum (i.e. EVM) and all the newly launched SC must be tested using this environment. This would take care of SC which are launched without testing, and b) Many good tools have been developed by academia and it would be a good approach to incorporate them on the Remix website as plug-ins.

- For Solidity researcher: Solidity interpreter needs more improvements to catch the vulnerabilities. For catching mathematical errors, Solidity can incorporate a solver. This addition can also pave the way for detection of reentrancy error, which occurs due to the misplacement of account deduction statement.

- For general researcher: Improving and developing new tools and techniques is important. Some examples areas where tools can be developed are: i) to catch new vulnerability patterns such as reported in [11], [40], ii) to make the use of libraries safe for SC, and iii) design strategies to reduce the miner’s time to generate a block that is currently 900 seconds.

- For Security Researchers: It is also important to develop more effective security testing and adequacy criteria that are unique for testing SC [54]. It is also important to develop algorithmic [55] and machine learning [56], and deep learning [57] approaches for detecting vulnerabilities and security defects in SCs.

### ACKNOWLEDGMENT

This research work is supported by National Science Foundation (NSF) under Grant No: 1821560.

### REFERENCES

[1] H. Hasanova, U.-j. Baek, M.-g. Shin, K. Cho, and M.-S. Kim, “A survey on blockchain cybersecurity vulnerabilities and possible countermeasures,” International Journal of Network Management, vol. 29, p. e2060, 2019.

[2] A. Alkhalifah, A. Ng, A. Kayes, J. Chowdhury, M. Alazab, and P. Watters, “A taxonomy of blockchain threats and vulnerabilities,” 2019.

[3] H. Chen, M. Pendleton, L. Njilla, and S. Xu, “A survey on ethereum systems security: Vulnerabilities, attacks, and defenses,” ACM Comput. Surv., vol. 53, 2020.

[4] W. Dingman, A. Cohen, N. Ferrara, A. Lynch, P. Jasinski, P. E. Black, and L. Deng, “Defects and vulnerabilities in smart contracts, a classification using the nist bugs framework,” International Journal of Networked and Distributed Computing, vol. 7, p. 121–132, 2019.

[5] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in Proceedings of the 6th International Conference on Principles of Security and Trust - Volume=10204. Springer-Verlag, 2017, pp. 164–186.

[6] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “Smartcheck: Static analysis of ethereum smart contracts,” in 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018. IEEE, 2018, pp. 9–16.

[7] K. Bhargava, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Beguelin, “Formal verification of smart contracts: Short paper,” in PLAS ‘16 Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, October 2016, pp. 91–96.

[8] Known attacks. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/

[9] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “Zeus: Analyzing safety of smart contracts,” in Network and Distributed System Security Symposium, 2018.
[55] ——. “Evolutionary algorithms for vulnerability coverage,” in 44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020, 2020, pp. 1795–1801.

[56] F. Abri, S. Siami-Namini, M. A. Khanghah, F. M. Soltani, and A. S. Namin, “Can machine/deep learning classifiers detect zero-day malware with high accuracy?” in 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019, 2019, pp. 3252–3259.

[57] N. Tavakoli, “Seq2Image: Sequence analysis using visualization and deep convolutional neural network,” in 44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020, 2020, pp. 1332–1337.