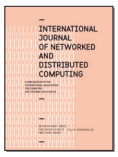
International Journal of Networked and Distributed Computing
Vol. 7(3); July (2019), pp. 121–132
DOI: https://doi.org/10.2991/ijndc.k.190710.003; ISSN 2211-7938; eISSN 2211-7946
https://www.atlantis-press.com/journals/ijndc



Research Article

Defects and Vulnerabilities in Smart Contracts, a Classification using the NIST Bugs Framework

Wesley Dingman[1], Aviel Cohen[1], Nick Ferrara[1], Adam Lynch[1], Patrick Jasinski[1], Paul E. Black[2], Lin Deng[1,*]

[1]*Department of Computer and Information Sciences, Towson University, 8000 York Road, Towson, MD 21252, USA*
[2]*Software Quality Group, Systems and Software Division, Information Technology Laboratory, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, USA*

ARTICLE INFO

Article History
Received 23 March 2019
Accepted 20 May 2019

Keywords
Blockchain
cryptocurrency
bug
smart contract
solidity
ethereum

ABSTRACT

Rising to popularity in the last decade, blockchain technology has become the preferred platform for the transfer of digital currency. Unfortunately, many of these environments are rife with vulnerabilities exploited by financially motivated attackers. Worse yet, is that a structured analysis and classification of these vulnerabilities is lacking. In this paper, we present the first formal classifications of these vulnerabilities using National Institute of Standards and Technologies Bugs (NIST'S) Framework and propose two new classes: distributed system protocol (DSP) and distributed system resource management (DRM).

1. INTRODUCTION

The blockchain is analogous to a distributed ledger of transactions that is programmed to record the transfer and storage of anything of value [1]. Each computer connected to the network in the system acts as a node, receiving a copy of the blockchain and functioning as an "administrator" on the network, continually verifying data and ensuring security within the platform. The fundamental principle behind this technology is that the distributed network it operates on minimizes the risk of a single vulnerability point - characteristic of a centralized database. While seemingly infallible, this technology has still been subject to exploitation by financially motivated attackers. The most famous instance, known as the DAO bug, occurred when an attacker utilized a "re-entrancy" vulnerability within an Ethereum smart contract that succeeded in stealing 60 million US$ [2]. For our research, we have decided to focus our attention on the Ethereum blockchain, presently the second most popular cryptocurrency with a current market valuation of roughly 13 billion US$ [3].

Ethereum smart contracts are written in Solidity, a high-level Turing-complete programming language [4,5]. Through an extensive literature review of previous research and other relevant resources, we have collected, analyzed, and categorized existing bugs in smart contracts on the Ethereum blockchain. This work has led to the creation of a comprehensive master list of well-documented vulnerabilities. Upon compiling the master list, we inspected the bugs to determine their causes, attributes, and consequences. Having analyzed the bugs thoroughly, we began utilizing the Bugs Framework (BF) developed by the National Institute of Standards and Technology (NIST) to categorize known vulnerabilities within their established classifications [6].

Any technology that is involved in the transfer of currency will draw financially motivated attackers. The proliferation of vulnerabilities within blockchain technologies is primarily due to their limited documentation and classification within a comprehensive, self-contained system. Our aim is to be the first to provide a precise, thorough categorization of known bugs within a blockchain environment in the hopes of mitigating financial losses for users of the technology.

In summary, the contributions of this paper are:

(1) We provide a formal, comprehensive, and unified reference for the causes, attributes, and consequences of known vulnerabilities in smart contract systems for software developers;
(2) Our research findings can improve the overall quality of smart contract systems and increase user satisfaction;
(3) This research provides a contribution that shall help achieve a more stable and reliable blockchain environment for human society;
(4) Our research discoveries led to the proposal of two new classes - Distributed System Protocol (DSP) and Distributed System Resource Management (DRM) - due to bugs that fall outside the current scope of the BF.

The content of this paper provides supplemental information on the Ethereum blockchain, the NIST BF, the process we followed in our research, the analysis and classifications of known bugs and finally the proposal of the DRM and DSP classes for addition to

*Corresponding author. Email: ldeng@towson.edu

the NIST BF. The paper is organized as follows: Section 2 provides background information; Section 3 details our research methodology; Section 4 lays out our classification of various bugs within the NIST BF; Section 5 discusses new types of bugs which do not fall into the existing BF; Section 6 presents related works on the matter; Section 7 concludes our paper.

2. BACKGROUND

To focus the scope of our research, we begin with a synopsis of smart contracts, introduce information regarding the Ethereum blockchain and Solidity, and discuss the NIST BF.

2.1. Smart Contracts

Smart contracts are programs residing on a blockchain which have their correct execution enforced by the consensus protocol [2]. These contracts have the ability to encode a diverse set of rules presented in a programming language. These rules are typically enforced when executing transactions between users or other smart contracts. The contract in Figure 1 represents a smart contract that acts as a wallet and provides the functionality to withdraw funds. Recently, smart contracts have been criticized for various malfunctions that have occurred upon implementation, such as including unexpected executions or unintentionally locking away thousands of dollars in virtual currency [7]. Unlike classical distributed systems that can be patched when bugs are detected, smart contracts are irreversible and immutable once deployed to a blockchain [2]. Due to this, a great deal of consideration must be placed on the development of a smart contract before its deployment to a blockchain. Figure 2 shows a contract which is detrimental to itself and any other contract that relies on it for state information.

```
contract AWallet {
   address owner;
   mapping (address => uint) public outflow;
   function AWallet() { owner = msg.sender; }
   function pay(uint amount, address recipient) return (bool) {
      if (msg.sender != owner || msg.value != 0) throw;
      if (amount > this.balance) return false;
      outflow[recipient] += amount;
      if (!recipient.send(amount)) throw;
      return true;    }
}
```

Figure 1 | A simple smart contract representing a wallet [7].

```
pragma solidity 0.4.19;
contract Fund {
   mapping(address => unint) balances;
   function withdraw() public {
      if(msg.sender.call.value(balances[msg.sender])())
      balances[msg.sender] = 0;    }
}
```

Figure 2 | Contract containing "re-entrancy" vulnerability [8].

2.2. Ethereum and Solidity

Trailing Bitcoin, Ethereum, as of June 2019, is the second most popular cryptocurrency by market value [3]. Contrary to traditional distributed applications, Ethereum operates in a permission-less network where arbitrary participants can join [8]. Although these participants must adhere to a predetermined protocol within the Ethereum environment, there is still substantial opportunity for manipulation of the smart contracts that reside on the network. Subsequently, contracts that reside and operate on the Ethereum blockchain need to be aware of the semantics of the underlying platform to reduce the risk of manipulation. Several factors that cause the implementation of Ethereum smart contracts to be error-prone can be directly traced to Solidity. While appearing similar to JavaScript, Solidity executes many of its features in peculiar ways. Much of the vulnerabilities seem to be caused by a disconnect between the semantics of the language and the intuition of the programmers [7]. Also, the language fails to introduce constructs to deal with domain-specific aspects, like the fact that computational steps are recorded on a public blockchain and can be unpredictably reordered or delayed. The contract presented in Figure 2 contains a "re-entrancy" vulnerability where msg.sender can recursively call withdraw() and drain the funds before its balance is set to 0.

2.3. The Bugs Framework

In an effort to accurately and precisely define software vulnerabilities, NIST has developed the BF [6]. The BF includes rigorous definitions and static attributes of bug classes along with their related dynamic properties. The taxonomy of each BF classification is comprised of causes, consequences, and various code sites. Language independence, or the lack thereof, is also considered [9].

Through factoring and restructuring the developments of known bug repositories Common Weakness Enumeration (CWE), Software Fault Patterns (SFPs), Semantic Templates (STs), their precise and unambiguous bug classifications provide a unified approach for the community to analyze and categorize new vulnerabilities. Rather than creating new CWEs for every subtle nuance of a vulnerability, the vulnerabilities can be categorized definitively, allowing researchers and programmers to have a useful avenue to evaluate similarities and differences. By utilizing NIST's new methodology for bug classification, we were provided a more general environment to examine and quantify vulnerabilities within a blockchain environment.

3. METHODOLOGY

To accomplish our goal of classifying bugs in Solidity, we first researched known bugs and vulnerabilities to gain an understanding of their nature; this allowed us to create an inclusive and comprehensive list of known bugs and vulnerabilities. We examined research publications within the past 5 years from conferences and workshops by ACM and IEEE, as well as significant journals in software engineering, computer science, networking, and security. The following keywords were utilized in our search: Blockchain, Ethereum, Solidity, Smart contract, Bugs, Vulnerabilities, Weaknesses, Secure practices. As smart contracts are relatively new, we also searched technology web pages, blogs, and white papers.

This search provided us with nine peer-reviewed publications related to Solidity smart contracts or the Ethereum blockchain, and the security thereof. We also found six web resources which gave lists of known vulnerabilities, including the official Solidity site. We chose to include these web resources in our study because they represent the combined knowledge of Solidity developers regarding best practices and security vulnerabilities.

With this information at hand, we created a consolidated master list of bugs and vulnerabilities. We did this by examining each bug and removing duplicates. We considered any two bugs as duplicates if they came from two different sources and described the same issue. Once our comprehensive master list was constructed, we organized the bugs further according to prior research by Tikhomirov et al. [8]. This places each bug into one of four categories: security, functional, developmental and operational.

We sought to classify these bugs according to NIST's BF. To do this, we used NIST's website to read their descriptions of existing bug classes [6]. We also read NIST's published journals on the BF [10]. A majority of the vulnerabilities that we discovered fell outside of the scope of any existing BF classes. These are primarily vulnerabilities that arose either due to the protocol of the blockchain environment or the transfer of resources within the blockchain, e.g., ether and gas transferred between contracts and users. Bugs that fell outside of the scope of the BF and did not arise due to the blockchain protocol or the transfer of resources were given a classification of *Other*.

We established two new classes to expand upon, and hopefully append to, the NIST BF: DSP and DRM. These classifications will be presented in a subsequent paper and hopefully appended to the BF.

4. FINDINGS

In this section, we briefly describe all the Solidity bugs we collected that fell into BF classifications, in terms of their causes, attributes, code examples (wherever possible), and consequences, and we categorize them into their respective BF classes. Also shown is our master list in Table 1, which contains these vulnerabilities.

4.1. Incorrect Arguments Class

The parameters that are passed as arguments to a function causes errors and therefore are incorrect [Incorrect Arguments (ARG)].

4.1.1. EC recover malformed input

Malformed input in contracts using the ecrecover() function causes the function to return left over data from the return area of memory [11].

Cause: When malformed input is entered into a contract using the ecrecover() function, the precompile does not signal a failure. If the contract receives some kind of malformed input, it can cause this method to return garbage.

Attributes: When malformed input is entered into the ecrecover() function, the function is not able to signal whether the program was successful or not correctly. This can lead to the function returning left over data that was present in the return area of memory.

Table 1 | Master bug list

Bug name	Classification	
	Category	BF class
Constant optimizer subtraction	Operational	ARC
Delegate call return value	Functional	ARC
Event struct wrong data	Functional	ARC
Floating point, unchecked division	Functional	ARC
Nested array function call decoder	Functional	ARC
Overflow/Underflow	Functional	ARC
EC recover malformed input	Operational	ARG
Public lib functions do not return nested arrays	Functional	ARG
Skip empty string literal	Functional	ARG
Stack size limit	Security	BOF
Progressing state based on external calls	Security	DSP
Race condition	Security	DSP
Transaction ordering dependence	Security	DSP
Unpredictable state	Security/Functional	DSP
Delegate call	Security/Functional	DRM
Ether lost in transfer	Security	DRM
Gasless send	Functional	DRM
Locked money	Functional	DRM
Lopping through externally manipulated mappings or arrays	Operational	DRM
Send fails for zero ether	Operational	DRM
Transfer forward all gas	Security	DRM
Tx.origin	Security	DRM
Uninitialized storage parameters	Security	INI
Owner operations	Functional	KMN
Costly loop	Operational	LOP
Dynamic allocation infinite loop	Operational	LOP
Array access clean higher order bits	Security	MAL
Clean bytes higher order bits	Operational	MAL
High order byte clean storage	Security	MAL
Optimizer clear state on code path join	Security	MAL
Optimizer stale knowledge about SHA-3	Functional	MAL
Optimizer state knowledge not reset for jumpdest	Security	MAL
Ancient compiler	Developmental	Other
Default visibility	Developmental	Other
ExpExponentCleanup	Operational	Other
External contract referencing	Security	Other
Incorrect Interface	Developmental	Other
Libraries not callable from payable functions	Functional	Other
One of two constructors skipped	Developmental	Other
Re-entrancy	Security	Other
TypeCast/Inference	Developmental	Other
Unexpected ether	Security	Other
Variable shadowing	Developmental	Other
Zero function selector	Security	Other
False randomness	Security	PRN
Timestamp dependence	Security	PRN
Identity precompile return ignored	Security	UCE
Token API violation	Developmental	UCE
Unchecked external call	Security	UCE

Consequences: Can be used by attackers to get data previously in the return location of memory that they should not be able to access.

4.1.2. Public lib functions do not return nested arrays

Public library functions that return nested arrays return the arrays with zero as all of its elements [11].

Cause: When a contract makes a call to a public function stored within a Solidity library that returns a nested array, the data being returned is lost and replaced with all zeroes.

Attributes: This bug only occurs in public library functions within Solidity, so any internal functions within the contract are safe. When compiled, the compiler does not correctly return nested arrays from public library functions. The compiler does not warn the user either, allowing the resulting code to be executable.

Consequences: If a public library function that returns a nested array is used within a contract, there will be data loss resulting in logical errors.

4.1.3. Skip empty string literal

If an empty string literal is passed in a function call as a parameter, it is skipped by the encoder when compiled, causing all of the parameters proceeding it to shift to the left by 32 bytes [11].

Cause: When an empty string literal is passed in a function call, it is skipped by the encoder. This causes the following function parameters to be passed incorrectly into the function being called by shifting all of the parameters to the left in memory by 32 bytes.

Attributes: The compiler's encoder encodes all data to the right of the fifth byte of a variables' data. When the string literal is passed over, the encoder instead shifts the other parameters to the left by 32 bytes. This causes the data to be incorrectly encoded.

Consequences: The function call data becomes corrupt.

4.2. Buffer Overflow Class

Memory allocated to a buffer is exceeded. This allows memory beyond the buffer to be altered [Buffer Overflow (BOF)].

4.2.1. Stack Size Limit

The call stack is exceeded, and an exception is thrown [7].

Cause: The call stack is utilized when a contract invokes another contract, or itself by means of this.f(), is bounded to 1024 frames. If this limit is reached, then a further invocation will throw an exception.

Attributes: A continual sequence of nested calls is generated to create an almost full call stack. An attacker contract typically generates these calls. Once the call stack is almost full, the vulnerable contract is invoked.

Consequences: If the vulnerable contract does not contain the correct exception handling, then the vulnerable contract will be attacked through this vulnerability when invoked[1]. Figure 3 shows an example of Stack Size Limit. The Governmental contract allows for "players" to make investments in the contract and the last one to make an investment without another investment occurring for 1 min receives the jackpot (half the investment total) and sends the remaining ether to the contract owner. The vulnerable contract titled "Governmental" is exploited by the malicious contract called "Mallory". The owner of the vulnerable contract in this instance is exploiting their contract for financial gain. Essentially, Governmental is a honey pot trap that collects ether and only pays out to the owner. Mallory exploits the vulnerability stack size limit by recursively calling herself and in turn growing the stack. The result of the stack being enlarged is the intentional failure of the send statements contained in the resetInvestment() function. This failure

```
contract Governmental {
    address public owner;
    address public lastInvestor;
    uint public jackpot = 1 ether;
    uint public lastInvestmentTimestamp;
    uint public ONE_MINUTE = 1 minutes;

    function Governmental() {
        owner = msg.sender;
        if(msg.value < 1 ether) throw;
    }

    function invest() {
        if(msg < jackpot/2) throw;
        lastInvestor = msg.sender;
        jackpot += msg.value/2;
        lastInvestmentTimestamp = block.timestamp;
    }

    function resetInvestment() {
        if(block.timestamp <
            lastInvestmentTimestamp + ONE_MINUTE) throw;
        lastInvestor.send(jackpot);
        owner.send(this.balance - 1 ether);
        lastInvestor = 0;
        jackpot = 1 ether;
        lastInvestmentTimestamp = 0;
    }
}
contract Mallory {
    function attack(address target, uint count) {
        if(0 <= count && count < 1023)
            this.attack.gas(msg.gas - 2000)(target, count +1);
        else
            Governmental(target).resetInvestment();
    }
}
```

Figure 3 | An example of stack size limit [7].

[1]This vulnerability was patched with a hard fork [4].

allows the balance of the Governmental contract to grow every time this attack is executed due to the actual winner not being paid. In order for the owner to siphon the pooled ether from Governmental the owner would just allow for another round to terminate correctly.

4.3. Arithmetic or Conversion Fault

Incorrect or faulty solutions are produced by software due to flawed primitive type conversion, or violations of either domain or range [Arithmetic or Conversion Fault (ARC)].

4.3.1. Constant optimizer subtraction

The optimizer converts numerical values into routines when it can conserve gas by doing so. Certain numbers can be incorrectly represented by these routines when converted [11].

Cause: The optimizer in Solidity converts different numbers in smart contracts into routines in order to conserve gas. This affects mathematical computations, specifically in regard to subtraction, since the compiled code runs mathematical computation between two routines instead of executing the mathematical operations with the actual numbers.

Attributes: Numbers can be represented by routines in compiled Solidity code. In order for the optimizer to convert a number into a routine, the number needs to be used as a constant throughout the contract. This causes the optimizer to convert it into a routine to save gas. If this number is represented in hexadecimal starting with 0xff and follows with a large number of zeroes, the number can be incorrectly converted into an incorrect routine representation.

Consequences: Performing operations on routines that have been improperly calculated can cause simple mathematical operations to yield unexpected or incorrect values.

4.3.2. Delegate call return value

If another function's call data contains 32 zero bytes when the delegate call function is executed, the delegate call function will return as false regardless of whether an exception is thrown [11].

Cause: At the lowest level, the delegate call function takes the call data of another function and returns a Boolean to determine whether or not the call was successful. The return value of the delegate call is simply the data interpreted as a Boolean. If the parameter function returns at least 32 zero bytes, the delegate call will always return false even if the call did not throw an exception.

Attributes: At the lowest level, Booleans are stored as integer values. Because the EVM does not check if a function returned a value that is correct but still "false" as a Boolean, it behaves as if the function was not successful. In other words, if the called function returns nothing, the delegate call will fail regardless of whether or not the value returned was correct. Figure 4 presents a contract with this vulnerability; the function getdelegated() contained in the contract would return a false Boolean despite it being of perfectly legal syntax and only being passed a SHA-3 hash of the value variable.

```
contract TestContract {
   uint value;
   function set (uint _value) external {
      value = _value;
   }
   function getdelegated() external constant returns (bool)
   {
      return this.delegatecall(bytes4(sha3("get()")));
   }
}
```

Figure 4 | Delegate call return value - an example of the ARC class [12].

Consequences: This bug would make contracts fail that were otherwise written as they should. This function is extensively used in Solidity libraries and was discovered in an Ethereum client [12].

4.3.3. Event struct wrong data

If a struct is passed as a parameter to an event, the memory address of the struct will be stored in the Log rather than the data within the struct [11].

Cause: Events in Solidity are used to log whenever some sort of transaction or interaction with a contract takes place. When a struct is used within an event, incorrect data is logged. Instead of logging the data passed into the event, the event logs the address of the struct.

Attributes: Events in Solidity allow programmers to access a special data structure within Solidity called the Log. This structure stores data such as contracts' memory addresses and parameters passed into it through events. If a struct is passed into an event, instead of any of the data within the struct logged into the Log, only the memory address of the struct is logged.

Consequences: If a struct is passed into an event, the wrong data gets stored in the Log for the event. This could cause issues with transactions with that contract in the future.

4.3.4. Nested array function call decoder

Multi-dimensional, fixed-size arrays that are returned from a function have their data elements misinterpreted as memory pointers [11].

Cause: If a smart contract calls a function that returns an array that is multi-dimensional and has a fixed size, the elements of the array are misinterpreted as memory pointers.

Attributes: If the calling function accesses the elements of the returned array, that function can then interfere with different memory locations within the program. There is a regular expression within Solidity that will check if there are any functions that try to return a multi-dimensional, fixed-size array. However, it will not check whether that function is used or not.

Consequences: Accessing members of returned arrays can lead to memory corruption. This does not affect single-dimension fixed-size arrays, and the memory corruption can only occur when the function returning the array is called.

4.3.5. Overflow/underflow

Value limitations exist for integers, lack of consideration for these limitations can lead to manipulation and faulty results [11,13–15].

Cause: The Ethereum Virtual Machine (EVM) has integer data types that are designated with bit level specification; such as "uint8" for an 8-bit unsigned integer, or simply "uint," which is an alias for "uint256," a 256-bit unsigned integer. The bit level specification of integers causes value storage limitations. When performing addition, subtraction, or storing user input with integer variables that contain value limitations, overflow/underflow can occur.

Attributes: Since the current mathematical operators do not utilize the correct overflow/underflow safeguards, mathematical operation on integer data types can lead to overflow/underflow. Addition, subtraction, or user input that involves variables with value limitations, yet do not utilize value checking will allow overflow/underflow to occur. The contract presented in Figure 5 has a safe_add() function which utilizes the proper value checking on the result to ensure overflow has not occurred. This was done with the require() function, which shall throw an exception if the expression passed results in a Boolean value of false.

Consequences: Vulnerabilities such as overflow/underflow allows for exploitation to occur when unsafe mathematical operators are utilized on the various, bit level specified, integer data types present in Solidity. In order to mitigate these vulnerabilities the SafeMath library is recommended, as well the require() function, which allows value checking to occur.

4.4. Initialization Fault Class

An incorrect value is utilized, which was derived from the faulty initialization of a resource; the resource relates to any software construct that is utilized for the value(s) it contains [Initialization Fault (INI)].

4.4.1. Uninitialized storage parameters

In the event complex data types are uninitialized, unexpected complications can ensue based on where the references to the complex data types point in memory [13].

```
function add(uint value) returns (bool) {
    sellerBalance += value; //possible overflow
    // possible auditor assert
    // asert(sellerBalance >= value);
}
function safe_add(uint value) returns (bool) {
    require(value + sellerBalance >= sellerBalance);
    sellerBalance += value;
}
```

Figure 5 | Overflow/underflow - an example of the ARC class [14].

Cause: Uninitialized complex data types, such as structs, can lead to issues with storage pointers.

Attributes: Depending on the type of variable, the EVM will use memory or storage space for the variable data. If local storage variables are uninitialized, they can unexpectedly point to other local storage variables in the contract.

Consequences: Uninitialized storage variables can be used intentionally to exploit users. uninitialized storage variables could cause a contract to function differently than the developer had intended. Figure 6 shows an example of Uninitialized Storage Parameters. In this example contract, there is a vulnerability that effectively unlocks the initially locked contract. The unlocked variable is indirectly affected and can be changed due to the fact that newRecord is not initialized. Solidity stores state variables sequentially in storage, due to this unlocked will be stored in slot 0. Since Solidity defaults complex data types, such as structs, to storage when declaring them as local variables, it becomes a pointer to storage. Due to the fact that newRecord is uninitialized it is pointing to slot 0, where unlocked is stored. When setting newRecord.name to _name we are effectively changing the storage slot 0 where the variable unlocked is stored. If _name has its last byte be non-zero, then unlock is true and the contract is unlocked.

4.5. Key Management Bugs (KMN) Class

Cryptographic keys, and other pieces of information that perform the same functionality, are mishandled [Key Management Bugs (KMN)]. This results in a high possibility of information exposure.

```
// A Locked Name Registrar
contract NameRegistrar {
    // registrar locked, no name updates
    bool public unlocked = false;
    struct NameRecord { // map hashes to addresses
        bytes32 name;
        address mappedAddress;
    }

    //records who registered names
    mapping(address => NameRecord)
        public registeredNameRecord;
    //resolves hashes to addresses
    mapping(bytes32 => address) public resolve;
    function register(bytes32 _name, address _mappedAddress)
    public {
        // set up the new NameRecord
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;
        resolve[_name] = mappedAddress;
        registeredNameRecord[msg.sender] = newRecord;
        // only allow registrations if contract is unlocked
        require(unlocked);
    }
}
```

Figure 6 | An example of uninitialized storage parameters [13].

4.5.1. Owner operations

Owner privileges are not utilized, yet they are required for the state of the contract to change; this situation causes the contract to fail. [8,13,14].

Cause: These issues appear where owners have specific privileges in contracts and they must perform some duty for the contract to advance to the next state.

Attributes: If a privileged user, e.g., the owner, misplaces their private key or becomes inactive the entire contract will fail to operate. Effectively the contract relies on a single address possessed by the owner, this creates a single point of failure that is subject to human error. The example contract on display in Figure 7 is an Initial Coin Offering (ICO) contract that can demonstrate this type of Denial of Service (DOS) vulnerability. This contract requires the owner to finalize() the contract otherwise tokens cannot be transferred.

Consequences: Due to the reliance on the owner's unique privileges, the contract can experience a DOS, i.e., left in operable. The DOS will result in financial losses, for all who rely on the contract for financial compensation which is contingent upon on the state transition of the inoperable contract.

4.6. Memory Allocation and Deallocation Bugs Class

In the event that memory is allocated, or deallocated, the data that is contained in that piece of memory is mishandled [Memory Allocation and Deallocation Bugs (MAL)].

4.6.1. Array access clean higher order bits

The compiler does not correctly clean the higher order bits of elements within an array if they were changed to be <32 bits long [11].

Cause: When elements in an array were changed to values that were <32 bits long, the compiler did not properly clear the higher order bits. Thus, on accessing the element, it appeared to have a different value than it was assigned.

Attributes: This bug would cause data corruption when array data was changed. This would lead to undesired behavior all around.

```
bool public isFinalized = false;
address public owner; //gets set somewhere
function finalize() public {
   require(msg.sender == owner);
   isFinalized == true;
}
// ... extra ICO functionality overloaded transfer function
function transfer(address _to, uint _value) returns (bool) {
   require(isFinalized);
   super.transfer(_to, _value);
}
```

Figure 7 | Owner operations - an example of the KMN class [13].

Consequences: An attacker could use this exploit to cherry pick input data to gain access to undesired code paths through an overflow.

4.6.2. Clean higher order bits on fixed byte array

Comparing two fixed byte arrays causes the compiler to convert their sizes to byte 32, causing their higher order bits to change, thus creating an incorrect comparison [11].

Cause: When two fixed byte arrays were compared, the higher order bits were considered in the comparison. Thus, values that were meant to be the same but were not considered equal.

Attributes: The type "byteNN" is simply just an array of bytes with the size of NN (from 1 to 32). During comparison, data would be viewed as if each of the fixed byte arrays were of size byte 32, and would just read in garbage data from memory.

Consequences: An attacker can use this bug to reach undesired code paths by entering specific input data.

4.6.3. Higher order byte clean storage

Higher order bytes are not cleared properly when reassigned, allowing for the data within the bytes to be rewritten during runtime [11,16].

Cause: Similar to above, higher order bytes for certain data types are not cleared properly when reassigned. All storage types are stored in blocks of 32 bytes, for some types the higher order bits were not cleared when assigning or editing data. An attacker could use a similar attack as in the previous bug to reached undesired execution paths.

Attributes: This bug represents a similar security threat to the last one. It is possible to use this bug to overwrite data that is stored in variables.

Consequences: This bug allows an attacker to write to storage variables during runtime. It affects every common data type in the language, so the Ethereum developers suggested people completely rewrite contracts [16].

4.6.4. Optimizer clear state on code path join

When the optimizer tried to join code paths, it would do so incorrectly leading to data corruption [11].

Cause: The optimizer computes equivalent code using a tree structure. When the optimizer has several code paths that it wants to merge into a single set of instructions, it uses the edges of the tree to do so. This calculation was done wrong.

Attributes: The optimizer is used to improve performance of compiled code. It does so by searching through compiled code for instructions that yield similar or equivalent results. When the optimizer would "rejoin" equivalent code paths, it would not properly reset back its intended position, leading to data corruption during runtime.

Consequences: An attacker would have to find specific ways that the optimizer would join individual instructions in order to use this bug to compromise a contract. To do this, that attacker would have to analyze the code at the instruction level, which would be very difficult.

4.6.5. Optimizer stale knowledge about SHA-3

The optimizer did not clear data that was used to calculate the hash of a block, resulting in incorrect hashes on blocks in the blockchain [11].

Cause: The optimizer saves data from previous instructions and analyzes that data to avoid recalculating values that are already known. Here, the optimizer did not properly clear data that was used for calculating the hash for the blocks, resulting in erroneous hashes in certain blocks.

Attributes: Here, this bug directly affects the hash that connects the blocks on the blockchain. A contract that is compiled with optimization enabled would not be able to properly join the chain because its hash was incorrect.

Consequences: New contracts trying to join the blockchain could get rejected due to the incorrect hash calculation.

4.6.6. Optimizer state knowledge not reset for jumpdest

The optimizer incorrectly calculates the reference for the "jumpdest" from one piece of code to the other [11].

Cause: The optimizer finds specific code paths that, at the bytecode level, have the same results, thus making compiled byte code more efficient. A "jumpdest" (i.e., jump destination) is simply a reference from one snippet of code to another, more efficient, snippet that has the same result. After its analysis is done, it must clear the data used for that block, if it does not, the next block will be corrupted. The optimizer was supposed to join multiple paths at the various jumpdests using a graph-like data structure. At the edges of the graph, the optimizer was intended to use the empty state to simplify joining the nodes of the graph together. This functionality was implemented incorrectly [11].

Attributes: If jumpdests are linked incorrectly, the hash of the block will be different than expected. This leads to corrupt contracts that either are added to the blockchain erroneously, or are unable to be added to the blockchain.

Consequences: When the optimizer calculated new code segments incorrectly, the hash of the contract would be corrupted, causing problems when it joined the chain.

4.7. Pseudo-Random Number Generation Bugs Class

Pseudo-randomness requirements are not fully met for the output produced by the software [Pseudo-Random Number (PRN)].

4.7.1. False randomness

Miners have access to the data within a block before it is added to the chain and can influence its data until it is added [7,13,14].

Cause: The Ethereum blockchain protocol requires participants (miners) to verify blocks before they are added to the chain. When verifying a block, miners have access to all of the data within said block. Miners also can influence certain attributes of a block, such as the timestamp of when a block is mined.

Attributes: Ethereum and Solidity contracts have no true source of entropy. Some ways that have been used to generate randomness are based on future blocks such as timestamps, block numbers, hashes or gas limit. These are not truly random because miners have influence on them.

Consequences: Contracts which rely on randomness are vulnerable to miner manipulation or influence. For example, consider a lottery contract in which users purchase a ticket and one lucky user wins the jackpot. This requires randomness to choose the winner. If a miner wants to increase their chances, they could see what the contract uses as a source of entropy and try to manipulate it for their benefit.

4.7.2. Timestamp dependence

Block timestamps are susceptible to manipulation, and as such can allow manipulations of other constructs that utilize the timestamp. [2,7,13].

Cause: A whole host of applications utilize time constraints to ascertain which actions are permitted or mandatory in a current state. Typically, time constraints are executed by using block timestamps. Block Timestamps have been used for a wide variety of applications: entropy for random numbers, locking funds for a period of time, and various state changing conditional statements that are time dependent.

Attributes: The miner who creates a new block-instantiation of a contract on the blockchain can determine the timestamp of the block within a certain degree. Since miners can adjust timestamps slightly, there exists the potential for nefarious use; this potential is contingent upon the way in which the block timestamp is utilized. If the block timestamp represents a determining factor in a financial transaction and the miner holds a stake in the contract, he then could have motive to manipulate the timestamp to gain an advantage. The example contract in Figure 8 contains this fault. If this contract collects enough ether then the miner would have incentive, and ample opportunity due to the use of the block timestamp, to gain an unfair advantage and acquire the funds.

Consequences: Since block timestamp manipulation can be used by miners, block timestamps have the potential to lead to contract vulnerabilities.

4.8. Unchecked Error Class

There is either no check, a wrong check, or a check that fails to signal whether there was an error when one exists [Unchecked Error (UCE)].

```
contract Roulette {
   uint public pastBlockTime; // Forces one bet per block
   constructor() public payable {} // initially fund contract
   // fallback function used to make a bet
   function() public payable {
      // must send 10 ether to play
      require(msg.value == 10 ether);
      // only 1 transaction per block
      require(now != pastBlockTime);
      pastBlockTime = now;
      if(now % 15 == 0) {
         msg.sender.transfer(this.balance); }
   }
}
```

Figure 8 | Timestamp dependence - an example of the PRN class [13].

4.8.1. Identity precompiled return ignored

The identity precompile is ignored and the contract is allowed to run without being able to access the identity contract [11].

Cause: Solidity has an identity contract that it uses to copy memory. Sometimes the call to this contract can fail due to the precompile check for it in a contract being ignored.

Attributes: This bug can be avoided on the public blockchain due to there being a way to ensure that they never fail. Private blockchains are susceptible to this bug.

Consequences: When the identity precompile is ignored, the program is allowed to run without having access to the identity contract. This could lead to errors occurring within a contract on a private blockchain when the contract is referenced. This results in loss of data.

4.8.2. Token API violation

ERC-20 [17] is a technical standard for Ethereum for implementing tokens. It defines a list of rules for Ethereum tokens. If the ERC-20 standard token interface is not met, then the contract will not be able to trade its currency with others [8].

Cause: Violation of the ERC-20 standard token interface by implementing functions with incorrect return types, parameters, or names. Or by failing to implement interface functions entirely.

Attributes: ERC-20 is de-facto standard Application Programming Interface (API) for implementing tokens transferable units of value managed by a contract. Exchanges and other third-party services may struggle to integrate a token that does not conform to it.

Consequences: Contracts that wish to provide users with a currency will find that their currency does not meet the standards for trade with other contracts. Figure 9 shows an example of Token API Violation. ERC-20 functions such as approve, transfer, and trasferFrom should not have exceptions (revert, throw, require, assert) thrown inside them. Library functions, especially third party library functions, may throw exceptions.

4.8.3. Unchecked external call

If a contracts state changes while calling the CALL opcode, then the contract can be manipulated by the contract that called it [2,7,13–15].

Cause: Using the call opcode without checking the return value.

Attributes: The CALL opcode is often used to transfer ether to an address. If the CALL fails it will not revert the contract but will return false. both the address.call() and address.send() methods use the CALL opcode. The example presented in Figure 10 is of a contract which does not check the return value of address.send(). This send represents a change in the contract state. paidout is set to true after the send is made, assuming that the winnings have been paid out. This can be abused if the addressee causes the send to fail. This sets paidout to true and the balance of the contract remains the same.

Consequences: If a contract's state is changed when the call opcode is used, then the state of the contract can be manipulated by the addressee of the call.

4.9. Infinite Loop Class

An error has caused a potentially infinite Loop (LOP).

4.9.1. Costly loop

Looping over a resource-intensive function can cause the contract to lose all of its gas [8].

```
function transferFrom (address _spender, uint _value)
returns (bool sucesss) {
require(_value < 20 wei)
// ...
}
```

Figure 9 | An example of the token API violation [8].

```
contract Lotto {
   bool public payedOut = false;
   address public winner;
   uint public winAmount;
   // ... extra functionality here
   function sendToWinner() public {
      require(!payedOut);
      winner.send(winAmount);
      payedOut = true;
   }
   function withdrawLeftOver() public {
      require(payedOut);
      msg.sender.send(this.balance);    }
}
```

Figure 10 | Unchecked external call - an example of the UCE class [13].

Cause: A loop containing a resource-costly function. This vulnerability could be the result of faulty development or malicious intent by a third party. In the event that a malicious party obtains the ability to manipulate a mapping or array, the vulnerability called "looping over externally manipulated mappings or arrays" will cause a Costly Loop to occur.

Attributes: Ethereum is a very resource constrained environment. Gas is the computational cost of performing operation, and contracts can only use however much gas the user sends with a transaction.

Consequences: If a costly loop drains the contract of gas then the contract will fail. Figure 11 illustrates an example of Costly Loop. If array. length is large enough, the contract containing this loop will exceed the block gas limit and transactions utilizing this contract will fail.

4.10. Wrong Operation Class

An incorrect operation occurred in the software [Wrong Operation (WOP)].

4.10.1. Floating point/unchecked division

Floating point data types are not supported in Solidity, due to this developers must develop their own floating point data types; if done improperly incorrect results shall develop [8,13].

Cause: As of Solidity v0.4.24, neither floating point or decimal types are supported. Since floating point representations cannot be directly represented with their own data type, they must be made with integer types in Solidity.

Attributes: In the development or use of floating-point representations these things were not considered and led to incorrect results: (1) integer division, in Solidity, will be rounded down; (2) order of operations; (3) high precision to low precision conversion should only be done following mathematical operation. The contract shown in Figure 12 will cause incorrect results to occur despite the mathematical calculations being correct; this is due to a lack of a proper floating-point data type.

Consequences: If the process of using integer types for floating point representations is done improperly, it can produce errors and vulnerabilities. These errors can be especially pernicious when the faulty result relates to financial transactions.

5. DISCUSSION

During our categorization of smart contract vulnerabilities, it became apparent that many could not be appropriately classified within the BF. First, the classification of *Other* in the master list

```
for (uint256 i = 0; i < array.length; i++)
{
   costlyF();
}
```

Figure 11 | An example of costly loop [8].

```
contract FunWithNumbers {
   uint constant public tokensPerEth = 10;
   uint constant public weiPerEth = 1e18;
   mapping (address => uint) public balances;
   function buyTokens() public payable {
      uint tokens = msg.value/weiPerEth*tokensPerEth;
      // convert wei to eth, then multiply by toke rate
      balances[msg.sender] += tokens;
   }
   function sellTokens(uint tokens) public {
      require(balances[msg.sender] >= tokens);
      uint eth = tokens/tokensPerEth;
      balances[msg.sender] -= tokens;
      msg.sender.transfer(eth*weiPerEth);     }
}
```

Figure 12 | Floating point/unchecked division - an example of the WOP class [13].

represents vulnerabilities that are developmental in nature and the BF is not concerned with issues of this sort. Second, the principle of scarcity is implemented on code execution and the vulnerabilities that arise due to this are unique and their traits are not covered by any existing BF classification. Finally, NIST's BF does not contain any classes inherent to distributed systems, their focus is currently on bugs that can occur in traditional programming paradigms.

Due to these unique circumstances, we propose the introduction of two new BF classes: DSP and DRM. DSP bugs occur in software that produces unexpected results due to the distributed system protocol. A blockchain must follow specific protocols to ensure proper consensus is reached and chain integrity is maintained. This protocol can cause a delay between transaction submission and code execution. DRM bugs occur when software malfunctions are due to poor resource management. Specifically, Ethereum uses "gas" to place a value on the cost-of-code execution. This resource must be adequately managed to ensure all code is executed. These classes are introduced to provide a foundation for categorizing vulnerabilities that occur from the inherent properties of a consensus-based distributed system and the computational resources consumed within that system.

Currently the DSP and DRM classes do not fit within the BF model. The BF's classifications provide high-level definitions in order to classify common vulnerabilities among multiple programming languages. With our focus on Ethereum, we felt that DSP and DRM contain bugs specific only to Ethereum and consensus-based blockchain environments, rather than classical distributed systems. The DSP and DRM classes would need to be expanded upon by contrasting their characteristics with vulnerabilities found in other distributed systems. After this analysis, thorough and formalized definitions of the DSP and DRM classes can be constructed and added to the BF. Full descriptions of these proposed classes will be developed in future work.

6. RELATED WORK

Our research aims to provide a comprehensive understanding of smart contract bugs. This section discusses related work done in regard to the analysis, categorization, and identification of bugs, defects, risks, and security vulnerabilities in smart contracts.

Inspired by real-world instances of attack, Atzei et al. [7] provide a survey that lists six types of attacks identified within Ethereum smart contracts, i.e., the DAO attack, king of the Ether throne, multi-player games, Rubixi, GovernMental, and dynamic libraries. Each type of attack is addressed by a comprehensive discussion. These attacks not only include Solidity issues, but also consist of EVM byte code, and inherent issues from blockchain technology itself. They confirm that one of the leading causes of the proliferation of vulnerabilities across the blockchain was the lack of some formal documentation on the bugs.

Li et al. [18] also conduct a survey to systematically examine security risks in blockchain systems. Real attacks in blockchain systems and related vulnerabilities are analyzed. The paper of Li et al. [18] provides detailed descriptions of various security tools and enhancements that have been previously developed, such as SmartPool, a mining pool system that attempts to subvert the 51% attack. Future research directions in the area of blockchain security are listed, including developing more efficient consensus algorithms, various techniques to address privacy leakage regarding decentralized applications, and the development of efficient data clean up and detection.

Tikhomirov et al. [8] categorize bugs in smart contracts into four classes: security, functional, operational, and developmental. SmartCheck, a static analyzer that can detect these bugs, is presented. The experimental evaluation results, which utilized 4600 real-world smart contracts, indicate that SmartCheck is very effective.

Focusing on two common vulnerabilities in Solidity's smart contracts: reentrancy vulnerability and transaction-ordering dependence, Mavridou and Laszka [20] introduce a new framework for writing more secure smart contracts called FSolidM. This helps developers to write smart contracts as finite state machines.

Bhargavan et al. [10] discuss the different ways people can create malicious Solidity scripts and how the vulnerabilities can be avoided by the developers of the initial contracts. Then, they outline a framework to analyze and formally verify Ethereum smart contracts using F*.

The paper by Luu et al. [2] provides a detailed overview of the vulnerabilities present in Ethereum smart contracts, by categorizing four main classes of security bugs, i.e., transaction-ordering dependence, timestamp dependence, mishandled exceptions and reentrancy. In addition, the Oyente tool is presented, which can discover security bugs.

The paper by Wohrer and Zdun [15] lists design patterns, which used Grounded Theory techniques, for Solidity smart contracts which can be used to avoid certain known vulnerabilities. Patterns such as Check-Effects-Interaction-Pattern, Emergency Stop (circuit breaker) Pattern, Speed Bump Pattern, Rate Limit Pattern, Mutex Pattern, and Balance Limit Pattern were discussed.

Grishchenko et al. [21] create a semantic framework for analyzing the bytecode produced by the Solidity compiler. This bytecode analysis provides instruction granular security analysis that allows security personnel and testers to analyze the output of the compiler and evaluate how the EVM operates. Since Solidity is a language of such high abstraction and with many semantic nuances that are unique to smart contracts, this framework is different from most others for less specialized programming languages. This framework allows for Solidity bytecode to be scrutinized at the instruction level, allowing for various vulnerabilities to be tested for and analyzed. In addition, this paper also gives its classification for the bugs present in Solidity. The classification has four categories: Call integrity, Atomicity, Independence of mutable account state, and Independence of transaction environment.

The paper of Destefanis et al. [22] provides a case study for the Ethereum blockchain "Parity" attack, occurring in November of 2017. The paper analyzes the source code and the library implemented, and discusses how recognized best practices could mitigate detrimental behavior in blockchain technology. The paper reviews the existing approaches in smart contract development and makes a call for blockchain specific software engineering practices.

The paper by Baliga [23] gives a general survey of various consensus protocols implemented in modern blockchain technologies and defines the consensus protocol as the mechanism within the blockchain that ensures a common, unambiguous ordering of transactions and blocks that is intended to guarantee the integrity and consistency of the blockchain across geographically distributed nodes. The paper also discusses the issues that lead to the failure of consensus protocol: blockchain forks, consensus failure, dominance, cheating, and poor performance.

7. CONCLUSION AND FUTURE WORK

As the use of digital cryptocurrency, blockchain technology, and smart contracts continue to grow in popularity, it is necessary to establish a mechanism to classify these potential vulnerabilities in a way that is accessible for the public to view and access. Because of the financial investments that have been made into the space, it becomes more and more likely that attackers motivated by financial gain will search for possible vulnerabilities in the system. Unlike conventional finance, the decentralized nature of cryptocurrencies require security professionals whose organizations have an invested stake in the technology to be open and transparent with each other about possible attacks or vulnerabilities. In this paper we use the BF, provided by NIST, to achieve this. The BF enables a standardized way of categorizing and classifying vulnerabilities that appear in blockchain and smart contract systems. We used existing documentation and previous work in blockchain security analysis to classify discovered vulnerabilities within the proper BF Class. Once categorized we analyzed each bug's attributes, consequences, causes, and security level and created a definition. We took steps to generalize the bugs to make them applicable to other technologies where possible, while keeping specific examples in Solidity and Ethereum.

Many of the bugs fit into already established classes in the BF. Those bugs that did not fit led us to create two new classes: the DRM and DSP classes. DRM contains bugs that can occur within a resource constrained environment, while DSP bugs can occur with many different distributed system environments. These classes varied from the classes already present in the BF. Due to this, these two classes and their contents need to be researched more to provide a comprehensive definition. For the future work, steps should be taken to generalize DSP and DRM as the current class is specific to Ethereum and Solidity. Research into other types of distributed

systems, other than blockchain, that contain bugs similar should be done to justify the reason for their presence in the BF.

Throughout our research of the issues that can appear in Solidity smart contracts, we have concluded that the Ethereum blockchain is a volatile and unpredictable environment which requires significant improvements to make it a reliable medium for digital transactions. Our research has allowed us to understand and classify the bugs on our master list. The formal, comprehensive classifications provided in this paper enable smart contract developers to deeply understand defects and vulnerabilities, which can further improve the overall quality and security of smart contract systems.

ACKNOWLEDGMENTS

This undergraduate student research project is supported via the Information Security Research and Education (INSuRE) project [24].

REFERENCES

[1] K. Christidis, M. Devetsikiotis, Blockchains and smart contracts for the internet of things, IEEE Access, IEEE, 2016, pp. 2292–2303.
[2] L. Luu, D-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), ACM, Vienna, Austria, 2016, pp. 254–269.
[3] Cryptocurrency Prices | Crypto portfolio tracker | Coin Stats.
[4] G. Peters, E. Panayi, Understanding modern banking ledgers through blockchain technologies: future of transaction processing and smart contracts on the internet of money, 2015, pp. 1–33.
[5] G. Wood, Ethereum: A Secure Decentralised Generalised Transaction Ledger, Ethereum Project Yellow Paper, 32 (2018) 1365–1367.
[6] NIST, NIST BF Welcome.
[7] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on Ethereum smart contracts (SoK), in: M. Maffei, M. Ryan (Eds.), Principles of Security and Trust. POST 2017. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2017, pp. 164–186.
[8] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, Y. Alexandrov, SmartCheck: static analysis of ethereum smart contracts, 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, ACM, Gothenburg, Sweden, 2018, pp. 9–16.
[9] I. Bojanova, P.E. Black, Y. Yesha, Y. Wu, The bugs framework (BF): a structured approach to express bugs, 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, Vienna, Austria, 2016, pp. 175–182.
[10] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, et al., Formal verification of smart contracts: short paper, 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS), ACM, Vienna, Austria, 2016, pp. 91–96.
[11] Solidity, Solidity v0.5.4 documentation.
[12] R.G. Schmidt, Enhanced-wallet is using return of delegatecall as return of functions it is calling #6201, 2017.
[13] V. Saini, HackPedia: 16 solidity hacks/vulnerabilities, their fixes and real world examples, 2018.
[14] Trail of bits. Not-so-smart contracts.
[15] M. Wohrer, U. Zdun, Smart contracts: security patterns in the ethereum ecosystem and solidity, 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), IEEE, Campobasso, Italy, 2018, pp. 2–8.
[16] C. Reitwiessner, Security Alert - Solidity - Variables can be overwritten in storage, 2016.
[17] ERC-20 Token Standard.
[18] X. Li, P. Jiang, T. Chen, X. Luo, Q. Wen, A survey on the security of blockchain systems, Future Gen. Comput. Syst. (2017).
[19] L. Luu, Y. Velner, J. Teutsch, P. Saxena, Smartpool: practical decentralized pooled mining, in: Proceedings of the 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 1409–1426.
[20] A. Mavridou, A. Laszka, Designing secure ethereum smart contracts: a finite state machine based approach, 22nd International Conference on Financial Cryptography and Data Security (FC), 2018.
[21] I. Grishchenko, M. Maffei, C. Schneidewind, A semantic framework for the security analysis of ethereum smart contracts, in: L. Bauer, R. Küsters (Eds.), Principles of Security and Trust. POST 2018. Lecture Notes in Computer Science, Springer, Cham, 2018, pp. 243–269.
[22] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, R. Hierons, Smart contracts vulnerabilities: a call for blockchain software engineering?, 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), IEEE, Campobasso, Italy, 2018, pp. 19–25.
[23] A. Baliga, Understanding Blockchain Consensus Models, Persistent, 2017.
[24] A. Sherman, M. Dark, A. Chan, R. Chong, T. Morris, L. Oliva, et al., INSuRE: collaborating centers of academic excellence engage students in cybersecurity research, IEEE Security & Privacy, IEEE, 2017, pp. 72–78.