2020 7th International Conference on Dependable Systems and Their Applications (DSA) | 978-0-7381-2422-3/20/$31.00 ©2020 IEEE | DOI: 10.1109/DSA51864.2020.00031

Artemis: An Improved Smart Contract Verification Tool for Vulnerability Detection

Anqi Wang
School of Computer Science and Engineering
Beihang University
Beijing, China
wanganqi@buaa.edu.cn

Hao Wang
School of Computer Science and Engineering
Beihang University
Beijing, China
wangritian@buaa.edu.cn

Bo Jiang*
School of Computer Science and Engineering
Beihang University
Beijing, China
gongbell@gmail.com

W. K. Chan
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

***Abstract*—Smart contracts are programs running on top of the blockchain platform. With smart contract, users can use code to define arbitrary rules to manage their assets. However, the security vulnerabilities within those smart contracts have caused significant financial losses to their users. Existing smart contract verification tools based on symbolic execution are still not comprehensive in terms of the types of vulnerabilities detected. In this work, we present Artemis, an improved smart contract verification tool that can detect vulnerabilities that includes greedy, block information dependency, gasless send, and dangerous delegatecall. We have thoroughly evaluated our tool on 12899 smart contracts in terms of vulnerability detection effectiveness and efficiency. The experiment results show that the Artemis tool is precise and cost-effective for practical use.**

Keywords- Smart Contract Verification; Symbolic Execution; Ethereum; Vulnerability Detection

I. INTRODUCTION

The blockchain technologies are developing fast in recent years. The blockchain network was originally proposed as a network for value transfer [10]. Later, blockchain platforms supporting smart contracts [25] emerged, such as the Ethereum [21][22] platform. These smart contracts enabled blockchain platform usually supports a Turing-complete programming language, allowing developers to write smart contracts code to manage their assets. With the smart contracts as the backend, engineers can further design frontend interfaces to build Decentralized Applications (DApps). Smart contracts have enabled a wide range of DApps in practice, such as supply chain finance, instant messaging, games, insurance, etc. The Ethereum accounts (including both the smart contract account and external account) are managing 105 million of Ether, which is about 17.3 billion of USD in May 2019 [27].

However, managing so many assets with smart contracts naturally makes them lucrative targets for hackers [19]. Indeed, the vulnerabilities with smart contracts have led to huge financial losses to the end users. The infamous the DAO contract vulnerability [18] has resulted in $60 million US loss. The smart contracts of Parity wallet have suffered from two vulnerabilities [31][32] . The first one has resulted in the loss of $60 million, and the second one has frozen more than $150 million in terms of Ether.

In previous work, several smart contract verification tools were proposed to detect vulnerabilities within smart contracts. Among them, the Oyente [8] tool is a well-designed smart contract verification tool based on symbolic execution. However, the Oyente tool is limited in that it can only detect 4 types of vulnerabilities. Several types of vulnerabilities are still not supported by it. In particular, the Block Information Dependency [29], the Gasless Send [9], the Dangerous Delegatecall, and the Freezing Ether vulnerability are four important vulnerabilities still not supported by the Oyente tool.

The smart contracts with Block Information Dependency vulnerability contains value transfer operations that are dependent on the block information, which can be manipulated by miners within a short interval. The Gasless Send vulnerability contains expensive fallback functions, which will trigger exception on execution. When a malicious smart contract transfers ether to the vulnerable smart contracts, the exceptions triggered by the expensive fallback function will return the ether back to sender. As a result, the malicious smart contract can wrongfully keep ether belonging to the vulnerable smart contract. For smart contracts with Dangerous Delegatecall vulnerability, the function called by the delegatecall function can be manipulated by a malicious smart contract. As a result, the

*All correspondence should be addressed to Bo Jiang.

978-0-7381-2422-3/20/$31.00 ©2020 IEEE
DOI 10.1109/DSA51864.2020.00031

malicious smart contract can provide well-designed argument for delegatecall to call whatever function it provides. This vulnerability has led to the first round of attack to the smart contracts of Parity wallet, which resulted in the loss of $60 million worth of ether. Finally, for smart contract with Freezing Ether vulnerability, it can receive ether but it provides no way to transfer the ether to other accounts. In another word, these smart contracts will freeze any ether sent to them. This vulnerability has led to the second round of attack to the Parity wallet, which resulted in the frozen of $150 millions worth of ether [32]..

In this work, we propose Artemis, an improved smart contract verification tool that supports the detection of Freezing Ether, Block information dependency, Gasless Send, and Dangerous Delegatecall vulnerabilities. Our experiment on 12899 real world smart contracts shows that Artemis can efficiently identify 1371 smart contracts vulnerabilities with relatively high precision.

The organization of the remaining sections is as follows. In Section 2, we will present the 4 types of vulnerabilities detected by our tool. In Section 3, we will show the detailed vulnerability detection algorithms for each type of vulnerability. Then,in Section 4, we present a comprehensive experimental study to evaluate the effective-ness of Artemis. In Section 5, we will discuss related work followed by the conclusion in Section 6.

II. BACKGROUND

In this section, we will illustrate the four types of vulnerabilities detected by our tool with examples.

A. Freezing Ether

The Freezing Ether smart contracts can accept Ether but have no way to transfer Ether. Therefore, Ether sent to them is frozen, affecting the circulation of ether seriously. If the ether of a user is managed by a greedy smart contract, he/she has no way to use the Ether anymore. As shown in TABLE I , the smart contract DreamData has a payable function receiving funds in line 3. However, there is no single ether transfer operation within the whole smart contract. As a result, any Ether received by the DreamData contract is locked forever.

B. Block Information Dependency

The Block Information includes Block Number, Block Difficulty and BlockHash. Miners can manipulate the block information within a short interval. Therefore, an attacker may manipulate block information to control the vulnerable contract to transfer ether or win a lottery in his/her favor. In general, if some critical operations (e.g., ether transfer) are dependent on Block information, the block information dependency vulnerability may exist. As shown in TABLE II , the block information is used in line 6 to control the send operation in line 10.

TABLE I. FREEZING ETHER VULNERABILITY IN DREAMDATA CONTRACT.

Line	Contract Code
1	***contract*** DreamData {
2	***event*** Data(uint length, uint value);
3	***function*** () ***public payable*** { *//function receiving Ether*
4	uint result;
5	***for*** (uint i = 0; i < msg.data.length; i ++) {
6	uint power = (msg.data.length - i - 1) * 2;
7	uint b = uint(msg.data[i]);
8	***if*** (b > 10)
9(1)	result += b / 16 *
9(2)	(10 ** (power + 1))
9(3)	+ b % 16 * (10 ** power);
10	***else***
11	result += b * (10 ** power);
12	}
13	Data(msg.data.length, result);
14	}
15	}

TABLE II. BLOCK INFORMATION DEPENDENCY IN PPBC_ETHER_CLAIM CONTRACT.

Line	Contract Code
1	***contract*** PPBC_Ether_Claim {
2(1)	***function*** refund_deposits(string password){
2(2)	*//anyone can call this*
3(1)	***if*** (deposits_refunded)
3(2)	***throw***; *//already refunded, throw*
4	***if*** (valid_voucher_code[sha3(password)]==0)***throw***;
5	*//wait until everyone has claimed or claim period has start*
6(1)	***if*** (num_claimed >= total_claim_codes
6(2)	|| ***block.number*** >= ***2850000***){
7(1)	***for*** (uint256 index = 1;
7(2)	index <= num_claimed; index++){
8	bytes32 claimcode = claimers[index];
9	address receiver = who_claimed[claimcode];
10(1)	***if*** (!receiver.***send***(50 ether))
10(2)	***throw***; *//throw on error*
11(1)	valid_voucher_code[claimcode]
11(2)	-= 50 ether; *//deduct the deposit paid*
12	}
13(1)	deposits_refunded = true;
13(2)	*//Ensure the function is only used once*
14	}
15	***else throw***;
16	}
17	}

C. Gasless Send

A smart contract with gasless send vulnerability has an expensive payable fallback function. When Ether is transferred to the smart contract by Send function and there are no other functions can receive funds, the fallback function will be executed. And the upper bound of the gas consumption within the receiving function is only 2300 unit.

If the fallback function consumes more than 2300 gas, the transaction will fail with an exception and the Ether transferred in the transaction will be returned to caller.

The gas limit mechanism is originally designed to avoid reentrant invocation. However, an Attacker can claim that he has transferred Ether to contract with gasless send vulnerability to gain benefits such as rights and honors. But in fact the transaction was failed and the ether has been sent back to him. As shown in TABLE III , the fallback function in line 2 alone is too expensive as send operation in line 12 alone consumes 18410 worth of gas, which is over the default value 2300. When an attacker sends Ether to the contract, exception will occur during the execution of the fallback function. And the Ether will be returned to the attacker.

TABLE III. GASLESS SEND VULNERABILITY IN SQUAREROOTPONZI CONTRACT.

Line	Contract Code
1	***contract*** squareRootPonzi {
2	***function***() payable{
3	***if*** (msg.value == 1 finney) {
4	***if*** (this.balance > 2 finney) {
5	uint index =masterCalculator.length + 1;
6	masterCalculator[index].ethereumAddress=
6(2)	msg.sender;
7	masterCalculator[index].name=
7(2)	"masterly calculated: "
8	calculatedTo += 100 ether;
8(2)	*//which is short for number 1e+20*
9	masterCalculator[index].squareRoot =
9(2)	CalculateSqrt(calculatedTo);
10	***if*** (masterCalculator.length > 3) {
11	uint to = masterCalculator.length - 3;
12	masterCalculator[to].
12(2)	ethereumAddress.send(2 finney);
13	}
14	}
15	}
16	}
17	}

D. Dangerous Delegatecall

When a normal Call is used to call a function of another contract, the code is executed in the context of the called contract. In contrast, when Delegatecall is used, the code is executed within the context of the caller. In general, the purpose of using Delegatecall is to improve code reuse by implementing library functions. In some cases, a vulnerable smart contract may call delegatecall while manipulating its arguments. As a result, an attacker can call any function as long as he provides the signature of the function through arguments. As shown in TABLE IV , the argument "_data" in delegatecall at line 5 is controllable by the caller within the tokenFallback function.

TABLE IV. DANGEROUS DELEGATECALL VULNERABILITY IN STANDARD223RECEIVER CONTRACT.

Line	Contract Code
1	***contract*** Standard223Receiver {
2(1)	***function*** tokenFallback
2(2)	(address sender, address origin, uint value, bytes data) {
3(1)	tkn = Tkn(msg.sender, sender, ,
3(2)	value, _data, getSig(_data));
4	__isTokenFallback = true;
5(1)	***if*** (!address(this).***delegatecall***(_data)) return false;
5(2)	*//delegatecall with arguments controlled by caller*
6	__isTokenFallback = false;
7	***return*** true;
8	}
9	}

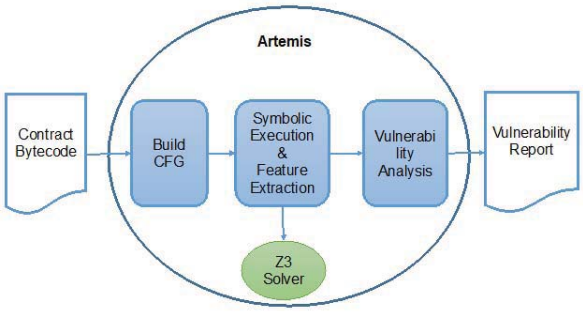

Figure 1. The Workflow of Artemis Tool.

III. THE DESIGN OF ARTEMIS TOOL FOR SMART CONTRACT VERIFICATION

The Artemis tool is built based on the Oyente symbolic execution framework. To support the detection of new types of vulnerabilities, we mainly extend its vulnerability detection module (as well as extracting some new features) to support the analysis of new vulnerability patterns. In this section, we present the vulnerability detection algorithm for each new type of vulnerability.

A. The Workflow of Artemis Tool

The workflow of Artemis tool is shown in Figure 1. In general, the input of the Artemis tool is the bytecode of the smart contract. Based on the contract bytecode, the Artemis tool will first build the Control Flow Graph (CFG) of the corresponding smart contract. Then, the Artemis tool begins to perform symbolic execution on the control flow graph starting from the first block of each function. During the symbolic execution, the important features (e.g., opcode sequence, path condition related to money flow, function invocation, block information usage, gas consumption, etc.) useful for vulnerability detection are collected along the way. Meanwhile, the symbolic execution engine will consult the constraint solvers (e.g., Z3) to check the satisfiability of the path constraints. Then, vulnerability analysis is performed

during and after the symbolic execution process based on the features collected. Finally, vulnerability analysis reports are generated based on the vulnerability analysis results.

Oyente has already supported detecting several types of vulnerabilities (e.g. Reentrancy vulnerability), so Artemis inherits the corresponding abilities and does not focus on the repetitive work. In this work, to detect the 4 new types of vulnerabilities, the Artemis tool mainly enhanced the vulnerability analysis module as well as feature extraction module. In the section that follows, we mainly discuss the vulnerability detection algorithm, which in general combines the core logic of both feature extraction and vulnerability analysis.

B. Detecting Freezing Ether Contracts

As shown in TABLE V , the input of the vulnerability detection algorithm is disasm file, which consists of opcodes disassembled from the bytecode of the contracts. In general, if a contract can receive ether but cannot send any ether, we consider it as Greedy. To check whether a contract can send any ether, we check whether there is any send related opcode (i.e., CALL, DELEGATECALL, CALLCODE, and SUICIDE) within the contract. To determine whether a contract can receive ether, we check whether the contract contains one or more payable function. Only those functions with payable modifier can receive Ether. Moreover, we call a function without payable modifier as "non-payable function". Since the payable modifier is introduced into Solidity since version 4.0, our detection algorithm is only applicable to those contracts using payable modifier.

Since the payable modifier detection method is different between ordinary functions and fallback function, we must divide a contract into 3 types at line 7 with the check_fallback_type function, which basically checks opcode patterns to determine the type of the contract. We omit its detailed implementation due to space limitation. Type 0 represents the contract without the fallback function; Type 1 represents a contract with a non-payable fallback function; Type 2 represents a contract with a payable fallback function. For Type 0 and Type 1, we count the total number of functions and the number of non-payable functions to determine whether the contract can receive Ether (line 8 to line 18). At line 10, we use a loop to iterate the opcodes in the disasm file. At line 11 and line 12, we use the combination of "PUSH4、EQ" opcodes to check the existence of a function. At line 13, we use the combination of "CALLVALUE、ISZERO" opcodes to check the existence of a non-payable function. Then the algorithm checks whether the number of non-payable function is smaller than the total number of functions at line 15. For contracts of Type 2, we can directly determine that the contract can accept Ether (line 19 and 20).

C. Detecting Block Information Dependency Contracts

To detect Block Information Dependency, we are checking whether the block information is used for ether transfer along the execution path. Using block information to change the storage of contract is not considered in this work because it is not as critical as malicious ether transfer. To detect these type of vulnerability, we first collect all path_conditions related to money flow during symbolic execution, which is already done within the Oyente framework. During symbolic execution, the Block Information is recorded as "IH_i", "IH_d", "IH_blockhash" corresponding to block number, block difficulty, and blockhash. If these types of Block Information are in money flow related path_conditions (which are recorded during symbolic execution), we mark the contract as vulnerable. As shown in TABLE VI , from line 3 to line 9, we traverse all the variables in the money flow related path condition to find the existence of block information opcode.

TABLE V. ALGORITHM FOR FREEZING ETHER VULNERABILITY DETECTION.

Line	Code
1	***def*** check_greedy_contract(disasm):
2(1)	send_instructions =
2(2)	['CALL','DELEGATECALL',
2(3)	'CALLCODE', 'SUICIDE']
3	***for*** i ***in*** range(0, len(disasm)):
4	instruction = disasm[i]
5	***if*** instruction[1] ***in*** send_instructions:
6(1)	***return False***
6(2)	*//Can send ether, not Freezing Ether contract*
7(1)	fallback_type = check_fallback_type(disasm)
7(2)	*//check the type of fallback*
8	***if*** fallback_type == 0 ***or*** fallback_type ==1:
9(1)	*/*Contracts without fallback function*
9(2)	*or with non-payable fallback function*/*
10	***for*** i ***in*** range (0, len(disasm)):
11(1)	***if*** disasm[i] == 'PUSH4'
11(2)	***and*** disasm[i+1] == 'EQ':
12	fun_num +=1
13	***if*** disasm[i] == CALLVALUE ***and*** disasm[i+1]
14	nonpayable_fun_num +=1
15(1)	***if*** (i==len(disasm)-1)
15(2)	***and*** (nonpayable_fun_num < fun_num):
16	***return True*** *//Have payable function, greedy*
17	***else***:
18	***return False*** *//No payable function, not greedy*
19(1)	***if*** fallback_type ==2
19(2)	*//Contracts with payable fallback function*
20	***return True*** *//Have payable fallback, greedy*

D. Detecting Gasless Send Contracts

Same as we do in 3.A, we first get the disasm file, which contains the opcodes of the contract. The check_gasless_send function is invoked upon symbolic execution of each block. Since the blocks of the fallback function are executed first during symbolic execution, we only need to setup a flag to notify when the detection can end. The gasless_send_flag is initialized to 1 to signify the detection is still in process. As shown from line 3 to line 4, block number 12 indicates that there is no fallback function in the contract while block number 13 indicates that the symbol execution has reached the first block after the fallback function (i.e., the fallback function has been

symbolically executed and checked.). So, the gasless_send_flag is set to 0 and the vulnerability detection process ends. The Ethereum Yellow paper has defined the gas consumption for each opcode. And the corresponding value are stored and updated in analysis["gas"] variable upon symbolic execution of each instruction. At line 5, we assign the current gas consumption value to current_gas_used from analysis["gas"]. If the detection is still in process and the current gas consumed is over 2300 in the current path, the gasless send vulnerability is reported.

TABLE VI. ALGORITHM FOR BLOCK INFORMATION DEPENDENCY DETECTION.

Line	Code
1	***def*** detect_bInfo_dependency():
2(1)	NUMBER_VAR = "IH_X"
2(2)	*//IH_X represents for IH_i, IH_d, or IH_blockhash*
3	***for*** cond ***in*** enumerate(path_conditions):
4	***for*** expr ***in*** enumerate(cond):
5	***if*** is_expr(expr):
6(1)	***if*** NUMBER_VAR in str(expr):
6(2)	*//block infor. matched in path condition.*
7	***return true***
8	***else***:
9	***return False***

TABLE VII. ALGORITHM FOR GASLESS SEND VULNERABILITY DETECTION.

Line	Code
1	gasless_send_flag = 1
2(1)	***def*** check_gasless_send(disasm):
2(2)	*//invoked upon symbolic execution of each block*
3	***if*** block == 13 ***or*** block == 12:
4	gasless_send_flag = 0 *//detection can stop*
5	current_gas_used = analysis["gas"]
6(1)	*/*analysis[“gas”] is updated upon*
6(2)	*symbolic execution of each instruction*/*
7(1)	***if*** gasless_send_flag == 1 ***and***
7(2)	current_gas_used >= 2300 :
8	gasless_send_flag = 0 *//detection can stop*
9	***rerurn True*** *//gasless send detected*

TABLE VIII. ALGORITHM FOR DANGEROUS DELEGATECALL VULNERABILITY DETECTION.

Line	Code
1	***def*** check_dangerous_delegatecall(disasm):
2	***for*** i ***in*** range(0, len(disasm)):
3	***if*** disasm[i] == 'DELEGATECALL':
4(1)	***if*** disasm[i+1]==‘ISZERO’ ***and*** disasm[i+2]==‘ISZERO’
4(2)	***and*** disasm[i+3]==“PUSH2”:
5	***continue***
6	***if*** disasm[i+1]==‘ISZERO’ ***and*** disasm[i+2]==‘PUSH2’:
7	***continue***
8	***else***:
9	***return True*** *//Dangerous DelegateCall Delected*

E. Detecting Dangerous Delegatecall

The algorithm to detect Dangerous Delegatecall vulnerability is shown in TABLE VIII If the delegatecall function is used in the contract, the DELEGATECALL opcode will appear in disasm file. Therefore, we first search the existence of DELEGATECALL opcode as shown in line 3 of our algorithm. If it exists, we will continue to check whether the function called is dependent on the message received. If the delegatecall function is used correctly, the specific signature of the called function will follow the DELEGATECALL opcode, which is usually passed through PUSH2 opcode. So if the opcode sequence is like “<DELEGATECALL, ISZERO, ISZERO, PUSH2>” or “<DELEGATECALL, ISZERO, PUSH2>”, the contract has used Delegatecall safely (line 4 to 7). In another word, these two opcode sequences mean that the DELEGATECALL specifies the function signature called, pointing to a specific function. Otherwise, the use of Delegatecall is vulnerable, as the called function is not specified, totally depending on the massage data provided by the caller (line 8 and 9).

IV. EXPERIMENT AND ANALYSIS

In this section, we report an empirical study to evaluate the effectiveness and efficiency of the Artemis smart contract verification tool.

A. Research Questions

We study two critical research questions as follows:

RQ1: Is Artemis effective to detect the four types of smart contract vulnerabilities?

For this research question, we want to compare Artemis with the MAIAN verification tool in terms of vulnerability detection effectiveness.

RQ2: Is Artemis efficient in terms of vulnerability detection?

The answer to this research question will help us understand the cost-effectiveness of the Artemis tool for practical use.

B. Experimental Setup

We use a desktop as our testing environment. The desktop is running Ubuntu 16.04 LTS and is equipped with Intel i5 4-core CPU and 8GB of memory. The versions of the tools and libraries used in the experimental environment are python v3.6.5, evm-1.7.2-stable, solc-0.55, z3-4.5.0, and geth-1.8.23.

- **Subject Programs.** We downloaded approximately 13,100 smart contracts with verified source code on Etherscan [23] between October 2017 and December 2018, which are parts of the all smart contracts offered by the site during this period. Finally, duplicate and invalid contracts were removed. The remaining 12899 smart contracts are used as our experiment subjects. We compile the source code of these smart contracts to get their bytecode. Then we use these bytecode of the contracts as the input for vulnerability detection. We choose these smart

contracts with source code because it is easier for us to manually verify the experiment results. But the Artemis and MAIAN tools only need the bytecode of the smart contracts to work.

- **Experiment Procedure.** Since Artemis is built based on the Oyente tool to detect more types of vulnerabilities, we borrow the default execution parameters of Oyente for Artemis to verify the smart contract bytecode. In particular, we set DEPTH_LIMIT and LOOP_LIMIT to 50 and 10, respectively. In fact, we have evaluated different combinations of values for DEPTH_LIMIT and LOOP_LIMIT with a subset of smart contracts. For DEPTH_LIMIT, we have tried 25, 50, 75. For LOOP_LIMIT, we have tried 10, 20, 30, 40, and 50. Finally, we found the default combination of 50 and 10 used in Oyente seems to produce the best vulnerability detection effectiveness.
  For the MAIAN tool, we also used the default parameters for vulnerability detection. Since MAIAN divides the risk of greedy vulnerability into two levels. We only select the one with higher accuracy. That is, the vulnerability detection result that reports the contract as "The code does not have CALL/SUICIDE/DELEGATECALL/CALLCODE thus is greedy!".

C. Results and Analysis

In this section, we present the results of our empirical study to evaluate the effectiveness and efficiency of Artemis tool

Vulnerability Detection Effectiveness of Artemis. In this section, we present the vulnerability detection results of Artemis.

TABLE IX. VULNERABILITY DETECTION RESULTS OF ARTEMIS.

Vulnerability Type	Number of Vulnerabilities	FP
Freezing Ether	80	4
Block Info Dependency	53	0
Gasless Send	1213	0
Dangerous Delegatecall	25	0

As shown in TABLE IX , each row shows the name the vulnerability, the number of vulnerable smart contracts detected by Artemis and the number of false positives (FP). Note that the false positives are determined through manual checking of each reported vulnerable contract.

For the freezing ether vulnerability, the Artemis tool flagged 80 smart contracts with 4 false positives. We examined the 4 smart contracts, and found that these contracts used the ".value" function to create a new smart contract and transfer ether to it. For example, in TABLE X , the contract called C2 contains a createAndEndowC1 function, which uses ".value" function to create C1 and transfer ether to it. The pattern of the opcodes therefore escaped from the detection logic of Artemis. We plan to fix this within our tool in future work.

TABLE X. SMART CONTRACTS WITH .VALUE FUNCTION TO TRANSFER ETHER ON CREATION.

Line	Contract Codes
1	***contract*** C2 {
2	…
3(1)	***function*** createAndEndowC1
3(2)	(uint arg, uint amount) ***payable*** {
4	*// Create and send the Ether*
5	C1 newC1 = (new C1).***value***(amount)(arg);
6	}
7	}

For Block Information Dependency, the Artemis tool detected 53 vulnerable smart contracts. We checked those smart contracts manually for confirmation. The contracts reported by Artemis contain no false positives. However, we did find a few false negatives (around 44 cases). We checked those false negative cases, where the block information is indirectly used for sending ether through complex data flow dependency. As shown in TABLE XI , the amount variable controls the ether transfer and is only data-dependent on *block.number*. Since Artemis currently still cannot track data flow dependency, the false negative is produced.

TABLE XI. A FALSE NEGATIVE CASE OF ARTEMIS ON BLOCK INFORMATION DEPENDENCY.

Line	Contract Codes
1	***contract*** DBank{
2	…
3(1)	***function*** buyCore(address _addr, uint256 _value)
3(2)	private {
4	bool isNewPlayer = determinePID(_addr);
5	if (invested[_addr] != 0) *{/// If you have investment*
6(1)	uint256 ***amount*** = invested[_addr]* r/100*
6(2)	(block.number-atBlock[_addr])/
6(3)	blocks_;
7(1)	if (***amount*** <= dbk_){
7(2)	*//Amount is data dependent on block.number*
8	addr.***transfer***(***amount***);
9	dbk_ -= amount;
10	}
11	}
12	}
13	}

Finally, for Gasless send and Dangerous Delegatecall, the Artemis tool detected 1213 and 25 vulnerable smart contracts, respectively. After manual checking, we found no false positives.

So in general, Artemis is effective to detect these 4 types of vulnerabilities with low false positive rate.

Comparison with MAIAN on Freezing Ether Vulnerability. In this section, we are comparing Artemis

with the MAIAN symbolic verification tool in terms of vulnerability detection effectiveness as shown in TABLE XII The Freezing Ether vulnerability is the common vulnerability that both MAIAN tool and the Artemis tool can detect. In total, the Artemis tool has detected 80 vulnerabilities while the MAIAN tool has detected 25 vulnerabilities. After manual examination of those vulnerabilities identified by the tool, we found that the Artemis tool has 4 false positives but has no false negatives.

TABLE XII. COMPARISON WITH MAIAN IN TERMS OF FREEZING ETHER VULNERABILITY.

	Total	FP	FN
Artemis	80	4	0
MAIAN	25	2	53

In contrast, after comparison with Artemis, the MAIAN tool missed 53 vulnerabilities (i.e., 53 false negatives). After analyzing the vulnerability detection logic of MAIAN, we found this may lies in its vulnerability detection logic. During symbolic execution, MAIAN will send ether to the smart contract under verification and check whether the transfer is successful based on the returned opcode (i.e., STOP/RETURN). However, some of these functions receiving ether contains require guard conditions, which are hard to satisfy by merely transferring ether to the target contract.

TABLE XIII. SMART CONTRACTS WITH GUARD CONDITIONS WHEN RECEIVING ETHER.

Line	Contract Codes
1	***contract*** KryptoGiftsMessages {
2	***address private*** admin;
3	…
4(1)	***function*** addMessage(***string*** txId, ***string*** userMsg)
4(2)	***payable*** external{
5	***require***(msg.sender == admin);
6	txIdMessages[txId] = userMsg;
7	}
8	}

As shown in TABLE XIII , the addMessage function in the KryptoGiftsMessages contracts contains a guard function, which requires that the message sender must be a predefined address specified by admin. However, within MAIAN, there is no logic to check and satisfy such condition when sending ether to the contract under verification. Therefore, the MAIAN considers this contract cannot receive ether but in fact it can. There are also other smart contracts with guard conditions to receive ether by specifying the amount to receive. Therefore, MAIAN misses all these contracts during detection.

The MAIAN tool also reported 2 false positives. We examined the contracts manually, and found that the MAIAN tool considered the path to transfer ether as unreachable. However, the path is in fact reachable, it seems the path conditions are too complex to solve correctly by MAIAN.

In summary, compared the MAIAN, the Artemis tool leads to much less false negatives, which is more effective to detect freezing ether vulnerability. The only limitation of Artemis is that it is only applicable to smart contracts supporting the payable modifier (i.e., contracts written since compiler version 0.4.0).

Vulnerability Detection Efficiency. As shown in Figure 2, for Artemis, the average vulnerability detection time for each contract is 11.23 seconds. The average vulnerability detection time per contract for MAIAN is 0.98s. Note MAIAN can only detect one type of vulnerability during each execution. To detect multiple vulnerabilities, we have to run MAIAN for multiple times. In contrast, the Artemis tool can detect 8 types of vulnerabilities (i.e., including the 4 types of vulnerabilities already detectable by Oyente). Considering the types of vulnerabilities detected with one execution, the vulnerability detection efficiency of Artemis is not far from MAIAN. Finally, the absolute vulnerability detection time of Artemis is small enough for practical use.

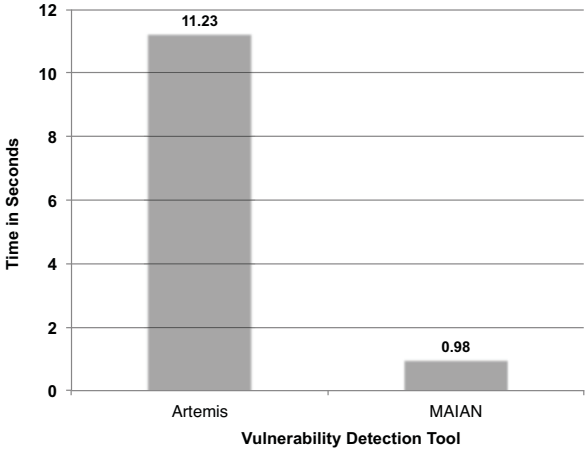

Figure 2. Vulnerability Detection Time Per Smart Contract.

D. *Threats to Validity*

In this study, we used around 12899 smart contracts with verified source code to evaluate and compare different vulnerability detection tools. These contracts are only a fraction of all smart contracts developed on Ethereum platform. We choose these smart contracts because they have source code to facilitate manual verification. A further experiment on vulnerability detection for more smart contracts will further strengthen the validity of our study.

Another threat to validity is the size the smart contracts. In general, most of the smart contracts crawled from Etherscan is small in size. We may further strengthen the validity of our study if we can collect more complex smart contracts projects from other sources such as Github for experimentation.

V. RELATED WORK

In this section, we review the closely related work on vulnerability detection of smart contracts.

A. *Symbolic Execution for Smart Contract Vulnerabilities Detection.*

Luu et al. proposed Oyente [8], a symbolic execution tool that provides three types of contract input: source code,

binary code, and remote input. It performs symbolic execution of contracts based on the control flow graph and detects several Ethereum contract vulnerabilities including the The DAO bug. The Osiris [14] tool improves Oyente to support the detection of integer overflow and underflow vulnerability. It also proposes to use taint analysis technique to improve the accuracy of the tool. Within their work, they also summarize the cause of integer overflow and underflow errors and put forward some best practices to write safe integer computation code. Mythril [26] uses symbolic analysis, taint analysis and control flow checking to detect security vulnerabilities of smart contracts, including integer overflow and underflow, suicide, Ether Thief, transaction order dependency, and others. Furthermore, their team comprehensively categorized smart contract vulnerabilities and test cases [30]. Nikolic et al. designed MAIAN [11], a symbolic execution tool to detect the greedy, the prodigal and the suicidal contracts through recording the invocation traces among contracts. They provided an accurate definition of the three classes of vulnerabilities and the triggering condition. Grossman et al. [3] focused on reentrancy attacks [13] and they defined the notion of Effectively Callback Free (ECF) objects. The contract with reentrancy vulnerability is not ECF. They implemented a dynamic monitor to verify the ECF of Ethereum. After each execution, it outputs all contracts that are not ECF.

ZEUS [7] uses symbolic execution method and abstract interpretation method to formally verify 20,000 contracts from Ethereum and Fabric. ZEUS contains three functional modules, including function development, source code compilation and source code verification. It performs static analysis on the smart contract and inserts the policy predicate as the assertion statement into the appropriate program point. ZEUS then uses its source code converter to convert smart contracts embedded with policy assertions into LLVM bitcodes. Finally, ZEUS calls its certifier to determine the assertion violation and determine if there is a policy violation. Securify [15] uses the security mode extracted from the semantic facts to locate errors in smart contracts. And the Securify language expressing security patterns and security properties are defined to get the security mode. For each property, compliance and violation patterns are verified through dependency graph.

B. Other Works on Smart Contract Vulnerabilities Detection.

Atze [1] provided a classification of vulnerabilities in smart contracts, and enumerates bugs such as calling unknown contracts, abnormal out-of-order, gasless send, type conversion, privacy protection, etc. They provide common programming traps in Solidity. Besides, they analyzed the vulnerabilities within smart contracts and the corresponding attack measures in their survey. Hirai et al. [4] proposed to formally verify smart contract with Isabelle, an interactive universal proof assistant that supports Higher Order Logic (HOL). The Eth-Isabelle project can produce a Coq definition of the Ethereum Virtual Machine, making it possible to verify smart contracts in Coq [5]. WHY3 [2][24] is an experimental formal verification tool, which relies on theorem provers to produce verification conditions. It allows Solidity contracts to be formally verified using Why3, solving the problem of integer overflow and divide-by-zero operations in contracts[12]. However, only a small subset of Solidity is supported for now. Clairvoyance[33] is a cross-function and cross-contract static analysis by identifying infeasible paths to detect reentrancy vulnerabilities. It supports fast yet precise path feasibility checking and yields remarkable detection accuracy.

In the fuzzing test of smart contracts, Jiang et al. released the ContractFuzzer [6], a precise and comprehensive fuzzing testing framework that can detect seven types of Ethereum smart contract vulnerabilities. ContractFuzzer generates fuzzing inputs based on the ABI [16][17]of smart contracts, defines test oracles used for detecting vulnerabilities, instruments the EVM to record the behavior of smart contracts at runtime, and analyzes running logs to report security vulnerabilities. Echidna [20] is another smart contract fuzzer detecting security problem by providing inputs of fuzz. It generates inputs according to actual code, and provides optional coverage guidance to find deeper bugs. Its limitation is that it does not provide a direct API to support smart contract security testing.

VI. CONCLUSION

Smart contracts are programs running on top of the blockchain platform to help managing our assets. However, managing assets with code will naturally attract hackers to exploit the vulnerabilities within the smart contracts for profit. Indeed, the security vulnerabilities within those smart contracts have caused significant financial losses to its users. With our preliminary study, we found existing smart contract verification tools based on symbolic execution are still not comprehensive in terms of the types of vulnerabilities to detect. Moreover, existing security fuzzing tool may still generate false positives and false negatives.

Therefore, in this work, we present Artemis, an improved smart contract verification tool that can detect vulnerabilities including greedy, block information dependency, gasless send, and dangerous delegatecall. We have thoroughly evaluated our tool on 12899 smart contracts in terms of vulnerability detection effectiveness and efficiency. Compared with existing verification tool, the Artemis tool in general has lower false negatives. The vulnerability detection efficiency of Artemis is also fast enough for practical use.

For future work, we will further enhance Artemis to support the detection of other new types of vulnerabilities. We also plan to study the integration of verification and fuzzing techniques to further improve the effectiveness of smart contract vulnerability detection. Finally, we will study how to generalize our tools on other popular blockchain platforms other than Ethereum.

ACKNOWLEDGMENT

The work is supported in part by the NSFC of China (project nos. 61772056), the GRF of HKSAR Research Grants Council (project nos. 11214116), the HKSAR ITF (project no. ITS/378/18), the CityU MF EXT(project no.

9678180), the CityU SRG (project nos. 7004882, 7005216, and 7005122).

REFERENCES

[1] Atzei, N., Bartoletti, M., Cimoli, T.: A Survey of Attacks on Ethereum Smart Contracts. In: The 6th International Conference on Principles of Security and Trust, Berlin, Heidelberg, pp.164-186. (2017).

[2] Filliâtre, J.C., Paskevich, A.: Why3: Where Programs Meet Provers. In: The 20th ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Heidelberg, pp.125-128.(2013).

[3] Grossman, S., Abraham, I., Golan-Gueta,G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. In: ACM Programming Languages, POPL, vol. 2, pp.28-48.(2018).

[4] Hirai, Y.: Formal verification of Deed contract in Ethereum name service. http://yoichihirai.com/deed.pdf, last access, 2016.

[5] Hirai, Y.: Ethereum Virtual Machine for Coq (v0.0.2).Retrieved June 12, 2018 from https://medium.com/@pirapira/ethereum-virtual-machine-for-coq-v0-0-2-d2568e068b18, last access, 2018.

[6] Jiang, B., Liu,Y., Chan,W.K.: ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: The 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). ACM, New York, NY, USA, pp.259-269.(2018).

[7] Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts. In:The 25¬th Annual Network and Distributed System Security Symposium (NDSS 2018), pp.18-21.(2018).

[8] Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making Smart Contracts Smarter. In: The 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). ACM, New York, NY, USA, pp.254-269.(2016).

[9] Miller,A., Warner,B., Wilcox,N., et al: Gas economics. http://github.com/LeastAuthority/ethereum-analyses/blob/master /GasEcon.md, last access, 2015.

[10] Nakamoto,S.: Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf , last access, 2009.

[11] Nikolic, I., Kolluri, A., Sergey, I., Saxena, P. Hobor, A.: Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In: The 34th Annual Computer Security Applications Conference (ACSAC '18). ACM, New York, NY, USA, pp.653-663. (2018).

[12] Reitwiessner, C.: Formal Verification for Solidity Contracts. https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts, last access, 2018.

[13] Sirer, E. G.: Reentrancy Woes in Smart Contracts. last access, 2019.

[14] Torres, C.F., Schütte, J., State, R.: Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In: The 34th Annual Computer Security Applications Conference (ACSAC '18). ACM, New York, NY, USA, pp.664-676.(2018).

[15] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: Practical Security Analysis of Smart Contracts. In:The 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). ACM, New York, NY, USA, pp.67-82. (2018).

[16] ABI of Ethereum Smart Contracts. https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI, last access, 2019.

[17] ABI specification of Ethereum smart contracts. http://solidity.readthedocs.io/en/v0.4.21/abi-spec.html#, last access, 2019

[18] Analysis of the DAO exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, last access, 2019.

[19] Announcement of imminent of hard work for EIP150 gas cost changes, https://blog.ethereum.org/2016/10/13/announcement-imminent-hard-fork-eip150-gas-cost-changes/ last access, 2019.

[20] Echidna. https://github.com/trailofbits/echidna, last access, 2019.

[21] Ethereum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper, last access, 2019.

[22] Ethereum Yellow Paper. https://ethereum.github.io/yellowpaper/paper.pdf, last access, 2019.

[23] Etherscan. The Ethereum Block Explorer. https://etherscan.io/, last access, 2019.

[24] Formal verification for solidity contracts. http://forum.ethereum.org/discussion//3779/formal-verfication-for-solidity-contracts, 2015, last access, 2019.

[25] Introduction to Smart Contracts. http://solidity.readthedocs.io/en/v0.4.21/introduction-to-smart-contracts.html, last access, 2019.

[26] Mythril. https://github.com/ConsenSys/mythril, last access, 2019.

[27] Number of Smart Contracts on Ethereum. https://etherscan.io/accounts/c. Last access, 2019.

[28] Oyente Analysis Tool for Smart Contracts. https://github.com/melonproject/oyente, last access, 2019.

[29] Predicting Random Members in Ethereum Smart Contracts. https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620, last access, 2019.

[30] Smart Contract Weakness Classification and Test Cases. https://smartcontractsecurity.github.io/SWC-registry/, last access, 2019.

[31] The Parity Wallet Hack Explained. https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7, last access, 2019.

[32] The Wallet Smart Contract Frozen by the Parity Bug. https://github.com/paritytech/parity/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol, last access, 2019.

[33] Ye, J.M., Ma, M.L., Lin ,Y.,Sui,Y.L.,Xue,Y.X.: Clairvoyance: Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In: The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020).