Intelligent Automation & Soft Computing
DOI:10.32604/iasc.2022.022801
Article

Tech Science Press

ISmart: Protecting Smart Contract Against Integer Bugs

Xingyu Zeng[1], Hua Zhang[1,*], Chaosong Yan[2], Liu Zhao[1] and Qiaoyan Wen[1]

[1]State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, 100876, China
[2]Olin Business School, Washington University in St. Louis, St. Louis, 63130, USA
*Corresponding Author: Hua Zhang. Email: zhanghua_288@bupt.edu.cn
Received: 19 August 2021; Accepted: 01 November 2021

Abstract: Blockchain technology is known as a decentralized, distributed ledger that records digital asset. It has been applied in numbers of aspects of society, including finance, judiciary and commerce. Ethereum is referred to as the next generation decentralized application platform. It is one of the most popular blockchain platforms that supports smart contracts. Smart contract is a set of codes that sored on blockchain and can be called and created as turing-complete programs running on the blockchain. Developers use smart contracts to build decentralized applications (Dapp) which has widely used cryptocurrency related project. As smart contracts become more popular and more valuable, they are faced with more risk of being hacked. As a result that smart contracts cannot be modified once deployed on the blockchain, it is a great challenge to fix and update deployed vulnerable contract which can lead to a huge loss of cryptocurrency and financial disorder. In this paper, we focus on Integer Bugs in Ethereum Smart Contracts and present ISmart, which protects deployed smart contracts against attacks caused by Integer Bugs. We implemented ISmart based on go-ethereum, a Ethereum client written in Go, by adding a simplified taint analysis component. In our preliminary, ISmart can prevent attacks accurately with little runtime overhead.

Keywords: Ethereum; smart contract; taint analysis; integer bugs

1 Introduction

Since Satoshi Nakamoto first set the Bitcoin and blockchain into motion in 2008 [1], the massive adoption of Bitcoin has fueled innovation. As a decentralized database, the blockchain allows transactions and data, to be stored and verified with no need of any centralized authority. With the development of blockchain, the emergence of Ethereum expands the function of bitcoin by Turing-complete smart contracts. The idea of smart contract was first proposed by Szabo in 1997 [2]. Ethereum provides a decentralized platform which can execute programs (smart contracts) by Ethereum Virtual Machine (EVM) [3]. More and more applications are deployed in blockchain. Liu et al. [4] proposed a food traceability framework based on permissioned blockchain. Wang et al. [5] stores electronic records on blockchain for secure provenance. Jiang et al. [6] proposed protocol for WLAN mesh security access

based on blockchain. Li et al. [7] protects energy trading based on Consortium Blockchain. Developers usually write smart contract code in a high-level language such as Solidity. Developers can create a new cryptocurrency with smart contract under an ERC Token Standard. In the real world, these cryptocurrencies are closely related to money which attracted attackers.

In April 2018, hackers attack the BecToken by integer bugs, leading to an extremely large amounts of tokens lost and the price of BEC tokens worthless. The code of vulnerable function is shown in Fig. 1. The line 3 of smart contract code indicates a multiplication without overflow limitation. If the result of multiplying is the **max** (max of 256-bit integer) + 1, the amount will overflow and be calculated to be 0 which will pass the constraint in line 5. Attackers can transfer unlimited BEC tokens.

```
//  BEC token

1 function batchTransfer(address[] _receivers, uint256 _value)
                    public whenNotPaused returns (bool) {
2     uint cnt = _receivers.length;
3     uint256 amount = uint256(cnt) * _value;
4     require(cnt > 0 && cnt <= 20);
5     require(_value > 0 && balances[msg.sender] >= amount);
6     balances[msg.sender] = balances[msg.sender].sub(amount);
7     for (uint i = 0; i < cnt; i++) {
8       balances[_receivers[i]] = balances[_receivers[i]].add(_value);
9       Transfer(msg.sender, _receivers[i], _value);
10     }
11     return true;
12   }
```

Figure 1: BEC vulnerable function

Academia proposed numerous different solutions to check smart contracts for vulnerabilities in static analysis, Oyente [8] is a symbolic execution tool which is used to find security bugs in smart contracts, Osiris [9] is a symbol execution tool based on Oyente, which focuses on handling integer errors, including overflow, symbol error and truncation error. ZEUS [10] verifies the correctness of smart contracts by using model checking. Due to the fact that the deployed smart contracts can't be changed, these static analysis tool can only give a suggestion before deployment but not deployed contracts. And limited to offline environment, these tools may not cover more bugs in complex path compared with runtime-protection.

The contributions of our work lay on the following aspects:

- We present ISmart, a simplified taint analysis component which protects deployed smart contracts against attacks caused by Integer Bugs.
- We analysis three different types of integer bugs arithmetic bugs, truncation bugs and signedness bugs and construct corresponding strategy in taint analysis.
- We run ISmart on real Ethereum smart contracts, and find that ISmart successfully interrupt dangerous transactions.

2 Background

2.1 Ethereum Virtual Machine (EVM)

Ethereum is referred to as the next generation smart contract and decentralized application platform. It consists of a network of mutually distrusting nodes form a decentralized public ledger. Users can create and execute smart contracts function by submitting transactions to Ethereum network. Miners will execute the smart contracts during the verification of blocks by EVM. Ethereum Virtual Machine (EVM) is a custom

virtual machine which executes smart contracts in Ethereum blockchain system which is a low-level stack-based virtual machine without register. Developer always uses Solidity to program smart contract application, which can be compiled to an instruction set of opcodes and bytecode further. Due to the smart contract will be executed by all miners in the network, Ethereum introduced the concept of gas to ensure miners benefits. Almost every execution of instruction will cost gas, when submitting a transaction, the sender has to provide the amount of gas that offered to the miner for the execution of the smart contract. If the gas is not enough, the transaction will fail because of out-of-gas exception. The success execution of a smart contract results in a modification of the world state σ. It is a data structure stored on the blockchain mapping an address a to an account state $\sigma[a]$.

2.2 GO- Ethereum

Go-Ethereum is one of the three original implementations (along with C++ and Python) of the Ethereum protocol written in Go. Go-Ethereum is used to execute transactions in Ethereum as showed in Fig. 2. When a transaction is accepted, it will be converted into a Message object by EVM. The Message object consists of four parts: ***from***, ***to***, ***amount*** and ***data***. ***from*** means the caller address of the transaction. ***to*** means that if the value of ***to*** is not null, EVM will find contracts code according to the value of ***to*** from ***StatedDB***, otherwise EVM will create new contracts according to transaction information and store related information into ***StateDB***. ***amount*** means the amount of value in transaction**.** ***data*** means the input of transaction which describes the called function in smart contract and the corresponding function parameter values. Then contracts code and input data will be sent to EVM interpreter. EVM is a stack based virtual machine. Four components need to be operated in the interpreter ***PC***, ***Stack***, ***Memory***, ***Gas***. ***PC*** is similar to the PC register in the CPU which points to the currently executed instruction. ***Stack*** is the execution stack which has a width of 256 bits and a maximum depth of 1024 bits. ***Memory*** is memory space. ***Gas*** represents a gas pool, the transaction will fail if it run out of the gas.

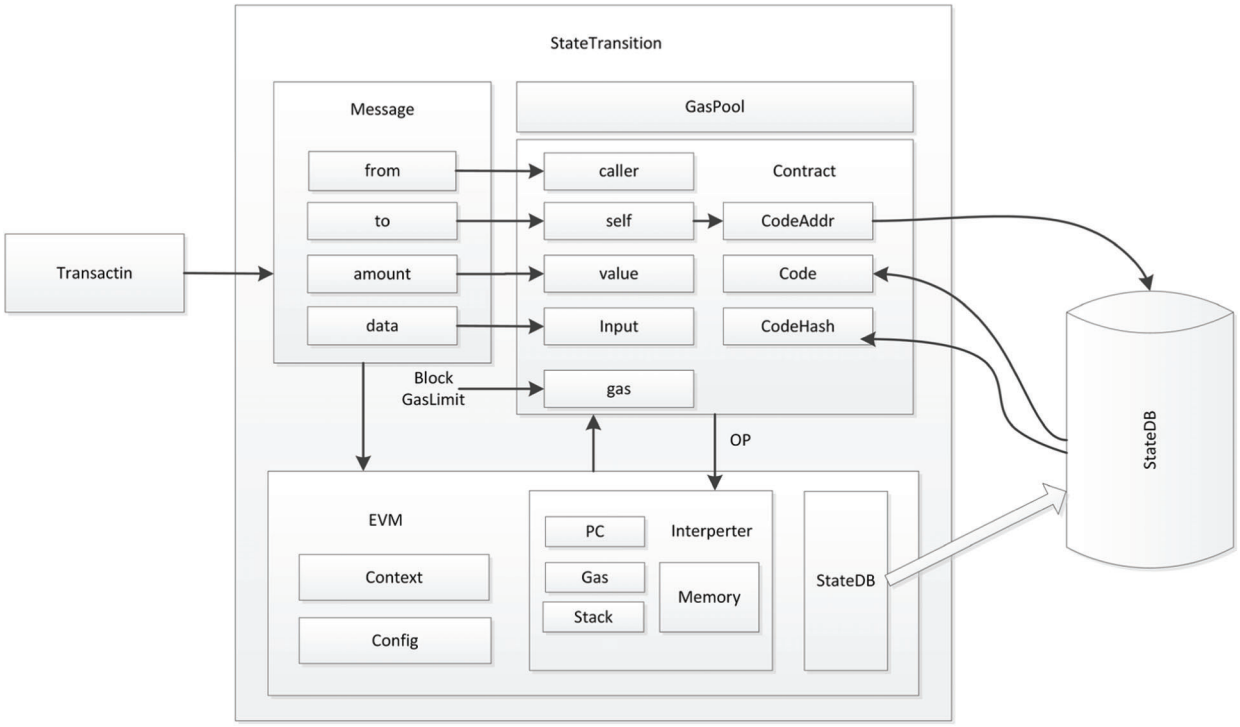

Figure 2: Overview of Go-Ethereum

2.3 Integer Bugs of Smart Contracts

In this section, we will briefly review on Integer Bugs of smart contracts.

Arithmetic Bugs. Arithmetic bugs include arithmetic overflow and arithmetic underflow. In addition, due to the special design of EVM, the result divided by 0 or module 0 is 0. This is controlled at the compiler level. The judgment of divisor 0 was not added before solidity 0.4.0. Arithmetic overflow is a value greater than the maximum value that can be stored, and arithmetic underflow is a value less than the minimum value that can be stored. As show in Fig. 3, for a 16-bit unsigned number, the maximum value that can be represented is $2^{\wedge}16 - 1$. For a value greater than $2^{\wedge}16 - 1$, it will be silently "wrap around".

```
//  Overflow bug

function add_overflow ( uint32 a, uint32 b)
    public returns ( uint ) {
        return a + b;
 }

function sub_overflow ( uint32 a, uint32 b)
    public returns ( uint ) {
        return a - b;
 }
```

Figure 3: An example of Overflow bug

Signedness Bugs. Converting an unsigned integer type to a signed type of the same width may introduce a signedness error. This conversion may change a positive value to a larger negative value. As shown in Fig. 4, this function only allows the caller to withdraw ether from the balance of the smart contract. However, if the parameter value is negative, it may be forcibly converted to an unsigned integer, and the amount transferred would be a huge positive integer.

```
// Signedness bug
function signedness_bug (int amount ) public {

    msg.sender.transfer ( uint ( amount ));
  }
```

Figure 4: An example of Signedness bug

Truncation Bugs. When converting a larger value into a narrower type value, it may cause a truncation error. In traditional languages such as Java, upward transformation is allowed and downward transformation is not allowed. For example, assigning a 64-bit value to an 8-bit integer type is easy to cause truncation errors, resulting in the loss of digital accuracy. Of course, if the value size of 64 bits is exactly less than the maximum value of 8-bit integer type, no truncation error will be raised. As shown in Fig. 5, the type of balance is a 32-bit unsigned integer. ***msg.value*** defaults to a 256-bit unsigned integer. It may result in loss of accuracy if converting ***msg.value*** to 32-bit unsigned integer.

```
// Truncation  bug
mapping ( address => uint32 ) balance ;

function truncation_bug () public payable {
        balance [msg.sender ] = uint32 ( msg.value );
    }
```

Figure 5: An example of Truncation bug

3 An Overview of ISmart

An overview of ISmart describing its workflow is shown in Fig. 6. ISmart is based on extending a GO-Ethereum client. We mainly add a taint engine and protector in EVM interpreter. Taint engine labels the data at pre-defined source and monitor it changes in the execution of the program. Protector protects contract from attacks by pre-defined strategy. As shown in Fig. 6, EVM will get function and function parameters from input data. Then EVM will get smart contract code form stateDB by code address. The information will be sent to interpreter to run the program. When program running, we will analyze interpreter memory and stack to find integer bugs and interrupt the transaction.

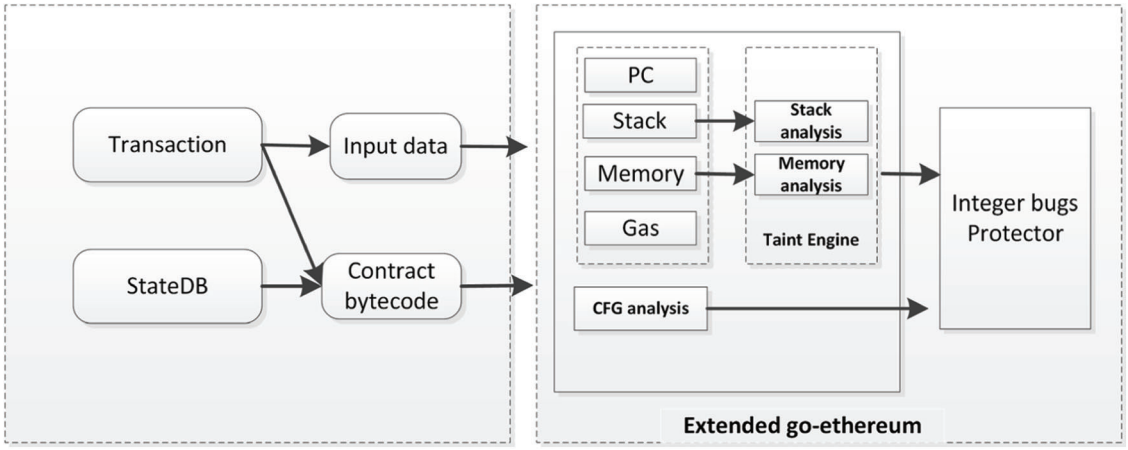

Figure 6: Overview of ISmart

3.1 CFG Analysis

When programming smart contract, developer may use some protection schemas to protect contract from attacks. They use constraints to determine whether an attack has occurred, such as ***required*** function or library so named SafeMath. ISmart needs to quickly detect the execution of transaction will be attached. ISmart will build CFG [11] from opcode to get the possible execution path. Due to that EVM is stack-based, only instruction conditional JUMPI and the unconditional JUMP can control the program execution process. We segment code to basic block which has not jumps instruction. When running detection, we can get run-time information from stack and memory so that we can simplify execution paths and easily get jump destination. We will analyze the instruction in current path to judge whether the protection mechanism should be triggered.

3.2 Taint Engine

Taint tracking is a popular technique in code analysis [12]. Taint analysis usually introduces the concept of taint source and sink. Taint source represents the direct introduction of untrusted data or confidential data into the system. Taint sinks are pre-defined points in the code, for example, ***add*** instruction may cause overflow bugs. ISmart will detect the taint if a tainted value reaches a taint sink.

Source. In the EVM, there are many of instructions that input and read data might cause integer bugs. We choose environment information and stack, memory, storage operations as the source in ISmart. Environmental information, such as ***calldatacopy*** and ***calldataload*** shown in Tab. 1, provides values at transaction level and attackers can set any value which is vulnerable. Operations such as ***sload*** or ***mload*** will read data from storage or memory which may be integer type data, so we analyze data type created by these operations. Ismart will record the set of storage addresses for detecting integer bugs.

Table 1: Source information

Instruction	Operation	Illustration
Calldatacopy	***Calldatacopy***(t, f, s)	Copy s bytes from calldata at position f to mem at position s
Calldataload	***Calldataload***(p)	Call data starting from position p (32 bytes)
Sload	***Sload***(p)	Storage [p]
Mload	***Mload***(p)	Mem [p… (p + 32))

Sinks. In the EVM, there are many of instructions that have impact on path execution, storage and token transfer. If the execution of the instruction is based on affected integer value, it may cause heavy financial losses. The operations such as ***sstore*** or ***jumpi*** have impacts on ***storage*** and ***stack.*** The parameters of the operations determine result. And some system operations such as ***call*** or ***return*** also change the execution path of the program. We choose ***sstore***, ***jumpi***, ***call***, ***mstore*** and ***return*** as sinks for our taint analysis.

3.3 Integer Bugs Protector

Integer bugs protector is the core component for protecting the transaction. The protector no only analyzes taint marks and values but also has a strategy to detect attacks. As shown in Tab. 2, we have developed different strategies for different vulnerabilities. For example, we mark the type of variables by specific instructions in stack to detect truncation error. They some public library, like SafeMath, that will protect the smart contract by protection scheme. They ensure right result of calculation by ***require*** or ***assert*** function. Integer bugs protector needs to quickly distinguish the success of attacks finally. We mark special operation, such as revert, in CFG struct to screen the potential of success attacks.

Arithmetic Bugs. In EVM, instruction ***add*** and ***mul*** may cause arithmetic overflow, and sub instruction may cause arithmetic underflow. In this case, it is necessary to define constraints on the boundary value. ***add*** and ***mul*** judge whether it is greater than the maximum value and ***sub*** judges whether it is less than the minimum value. For example, when two unsigned integers ***a*** and ***b*** are added, we need to checks arithmetic overflow constraint $(a + b > 2)^{\wedge}n - 1$, where n represents maximum bits of the two integers.

In addition, to detect the problem of dividing by 0 or modulus 0, check constraints need to be contracted. The division and modulus operators of EVM are signed/unsigned division (i.e., ***sdiv*** and ***div***) and signed/unsigned modulus (i.e., ***smod***, ***mod***, ***addmod*** and ***mulmod***). For example, for signed division, check whether the divisor can be zero. If the solver can meet the constraints under the current path conditions, there may be arithmetic errors.

Truncation Bugs. To find truncation errors, we should identify signed and unsigned integers first. In solidity, ***and*** and ***signextend*** are used to truncate signed and unsigned integers. The constraint for the ***and*** and ***signextend*** instructions is that the input value is greater than the output value. We need to filter out other type (i.e., ***address***) by conversion Schema. For example, the conversion of the type ***address*** is equivalent to 160 bits.

Signedness Bugs. To find Signedness errors, we identify the symbol of the integer value. Signed/unsigned type information about all integer values needs to be reconstructed from the EVM instruction executed. We mark all integer in three types (***signed***, ***unsigned***, ***unkown***). Lable ***signed*** means that the value is consider as singed integer the same as that lable ***unsigned*** means that the value is consider as unsinged integer. Lable ***unkown*** means that the value needs to juge by related instructions. When evm running, any integer is marked both as signed and unsigned type, the protector will trigger a Signedness error.

Table 2: Integer bugs detection

Type	Constrain	Related instruction
Arithmetic Bugs	$x\ op\ y \in [-2^*n, 2^*n-1]$	***add*, *sub*, *mul*, *div*, *mod***
Truncation Bugs	Same bits & (input = output)	***and*, *signextend***
Signedness Bugs	Only marked as signed or unsigned	***slt*, *sgt*, *lt*, *gt***

Protection process. The overall protection process is shown in Algorithm 1. We monitor instructions and variables that we consider important. First, we build CFG by contract bytecode and analyze the instructions in the execution path and basic block to quickly detect whether the path is vulnerable. Then, opcode ***call*** is a start of a transactions, meanwhile, we construct three monitoring sets: ***vulnerable_set*, *sign_set*, *trun_set***. ***vulnerable_set*** is used to store variables which is stained. ***sign_set*** is used to store variables with singed lable. ***trun_set*** is used to store variables with its number of bits. When evm running, if current opcode is in ***arithmetic_interesting***, such as ***add***, parameter and result will be sent to ***arithmetic_constrains*** function. If the result is vulnerable, the effect variable will be added to ***vulnerable_set*** and CFG path will be checked to juge whether quickly interrupt the process. If the opcode is in ***truncation_interesting***, such as ***signextend***, variable will be labeled and added to ***trun_set*** after checking the variable type and its width. If the opcode is in ***signedness_interesting***, such as ***slt***, variable will be labeled and added to ***sign_set*** after checking the variable singed type. If ***trun_set*** and ***sign_set*** checked result by constrains is vulnerable, they enter the same process as arithmetic detection. Finally, we will check the opcode in ***sensitive_ops*** to ensure that the results are not affected.

Algorithm 1: Instrumentation process

```
Input : bytecode, constrains, Interrupt,
define_function (CFG)
define_function (arithmetic_constrains)
define_function (truncation_constrains)
define_function (signedness _constrains)
cfg = CFG(bytecode)
while evm.isrun() do:
    if evm.pc.op == call:
      vulnerable_set = new set()
      sign_set = new signedness()
      trun_set = new truncation ()
else if evm.pc.op in interesting_ops:
   swich evm.pc.op:
        case arithmetic_ interesting:
          if arithmetic_constrains(args):
            vulnerable_set.add(integer)
            if cfg.blockTest():
               revert()
        case truncation_interesting:
              juge_truncation(integer)
            if truncation_constrains(trun_set, integer):
               vulnerable_set.add(integer)
```

(Continued)

Algorithm 1 (continued)

```
                if cfg.blockTest():
                    revert()
        case signedness _interesting:
            juge_sing(integer)
            if truncation_constrains(sign_set, integer):
            sign_set.add(args)
            vulnerable_set.add(integer)
            if cfg.blockTest()
                revert()
else if evm.pc.op in sensitive_ops:
        if integer in vulnerable_set:
            revert()
evm.run(OP)
```

4 Evaluation

ISmart mainly protects EVM against integer bugs. We evaluate the effectiveness and performance of ISmart on the Ethereum private-net. In the test, we need to deploy the contract on the private network and construct inputs of smart contract to trigger the attacks. Fortunately, there are some static analysis tools can help us to collect contracts and trigger information. We collect smart contracts from EtherScan, known as a famous Blockchain Explorer, and programing by ourselves. We selected 75 contracts that covering all bugs to test the performance of ISmart.

4.1 Experimental Environment

We run ISmart and origin client on a server that equipped with Intel i7 8 core CPU, 512GB SSD and 16GB memory in Ubuntu 18.04 LTS. In order to simulate practical condition, we deploy all contracts on the private network and synchronize partial status of the main network by Ethereum full node. We prepared the corresponding data to trigger the function. As shown in Fig. 7, we should construct the input data of transaction. Input data is usually divided into two parts: The first four bytes are called 4-byte signature, which are the first four bytes of the keccak hash value signed by a function as the unique identification of the function. Followed data is the parameters provided to the called function. EVM push the 4-byte signature into the stack through the ***calldataload*** instruction, and then compare it with the functions contained in the contract. If it matches, call the ***jumpi*** instruction to jump into the code and continue execution. Based on above, we can construct customized input data to trigger attack to verify the protection of Ismart.

4.2 Performance

In this section, we present the results of our protection as shown in Tab. 3. The columns show the vulnerability type, the number of vulnerabilities used for the test, and its protection effect. The rows represent the results for each integer vulnerability type. In the experiment, we only focus on the situation when the vulnerability is triggered by transaction and we collect imitation transactions constructed by existing tools and ourselves manually. The number of transactions that trigger vulnerability is variable. The tested contracts contain common CVE contracts, such as mentioned BECtokens, and also programing by ourselves. In three integer vulnerability cases, ISmart successfully interrupt the dangerous transactions from executing when any vulnerability detected. We can conclude that ISmart has excellent protection capacity.

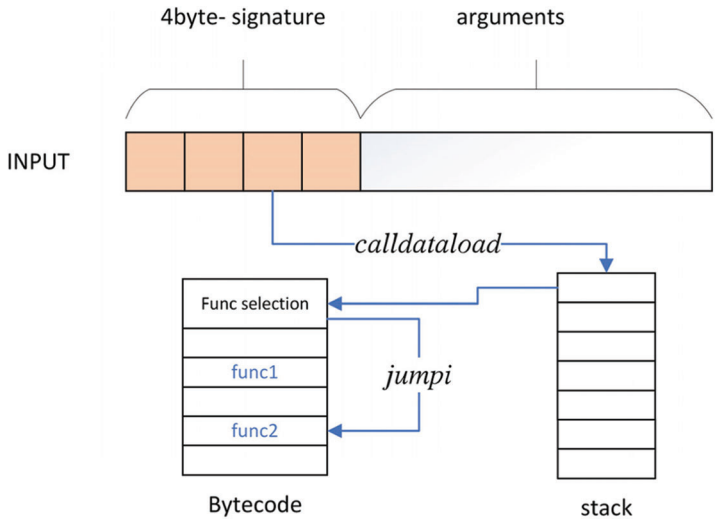

Figure 7: Smart contract transaction

Table 3: Performance of protection

Type	Amount	Interrupt
Arithmetic Bugs	30	100%
Truncation Bugs	25	100%
Signedness Bugs	20	100%

Due to that smart contracts run on distributed platform, Memory consumption and computational efficiency are particularly important. In Ethereum, all nodes will participate in the calculation of each transaction. Additional consumption ISmart casused will be attached to all nodes. As shown in Tab. 4, we tested ISmart and Origin-client memory and transaction efficiency. Official client needs 9243 MB memory in normal running, Ismart required on average 9455 MB of memory with integer bugs detection increased by 2.2% than official client. We also compare their execution efficiency through the number of transactions per second. Every transaction will call a function of smart contracts, we use partly vulnerable function by trigger information and normal function by synchronized information in mainnet. The average code length of function is 23 and the average transaction of official and ISmart are 13 and 11 respectively. The overhead is mainly due to recording data and strategy execution. Ismart caused extra costs when finding interesting opcode. There is no affect on normal execution of smart contract unless attack occurred. Compared with the origin-client, the extra costs are acceptable for potential financial losses.

Table 4: Overhead of protection

Type	Memory (averge)	Tx/s (averge)
Origin-client	9243 MB	13
Ismart	9455 MB	11

5 Related Work

Runtime Validation. Sereum [13] provides a tool to detect reentry attacks in real time. It extends the original Ethereum client to track the impact of each operation of transaction on storage and state. It identified re-entrancy attacks by taint analysis on modeled memory and state. Soda [14] is an EVM online detection platform which user can only find attacks but not avoid. It acts as a monitor for the full Ethereum node and trace execution information of the smart contract. If there is an attack detected through the information, it provides alarm information, but the illegal transaction will be handled by Ethereum finally.

Fuzzing test. ContractFuzzer [15] simulated Ethereum transaction execution model and collected sensitive data to trigger vulnerabilitie. It is running on the Ethereum network with a high execution rate, but because all parameters are generated randomly that can fail to cover deeper paths and expose vulnerabilities. Learning Fuzz [16] provides a machine learning model based on execution data produced by expert detecteor. Because it used the model to select the functions and parameters, it has a good performance in some cases. Machine learning also is applied in malware detection [17]. Wei et al. [18] provide a new fuzzing way in the detection of smart contracts. They combined Taint Analysis and Genetic Algorithms to find better execution parameters that may trigger vulnerability efficiently. REGUARD [19] uses fuzz testing to find reentrancy bugs in smart contracts. The detection of Reentrancy bugs is related to execution status which is efficiently in fuzzing test, but they inevitably miss critical vulnerabilities because of the random input.

Static Analysis. Researchers developed sound and scalable static analyzers [20–22] and formal verification tools [23–25]. Securify [20] is based on abstract interpretation to merges execution paths to avoid path explosion. ZEUS [10] takes a transform way that converts smart contracts source code into LLVM IRs by a defined policy. Then it uses the existing LLVM tools to detect the smart contract in convenient way. TeEther [21] is a symbolic execution tools that combines information flow detection with exploit generation. Benefit from Integration of information flow, it has a high precision. SMTCHECKER [22] is a verify tool developed by the Ehtereum Foundation to detect arithmetic bugs based on performing SMT-based bounded verification. Hirai [23] provides a way to prove safety of smart contracts in interactive theorem provers by formalizes the Ethereum Virtual Machine based on semantic analysis. They have a Semantic analysis. Grishchenko et al. [24] formalizes the Ethereum Virtual Machine by small-step semantics in F* framework. They use bytecode and constraint to define numbers of security properties of smart contracts. Lahiri et al. [25] provide a way that expresses high-level specification by state machine representation. They formulate formal specification and verification of smart contracts and verify whether the execution meets the specification.

6 Conclusion

Integer bugs are on of most commonly bug in Ethereum. We present the design and implementation of ISmart, a protector for preventing smart contracts from integer bugs. ISmart extends the origin Ethereum client with a taint engine and protector. We tested it in a real environment with various of smart contracts. ISmart accurately captures three types of integer vulnerabilities in the experiment. Moreover, in our evaluation, ISmart only increases memory consumption by 2.2% compared with origin client in high accuracy. Furthermore, we will try to implement our methodology on more EVM versions and detect more vulnerabilities.

Funding Statement: This research is supported by NSFC(Grant No.62072501), Key Project Plan of Blockchain in Ministry of Education of the People's Republic of China (Grant.2020KJ010802).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[2] N. Szabo, "The idea of smart contracts," 1997. [Online]. Available: http://szabo.best.vwh.net/smart.contracts.html.

[3] G. Wood, "Ethereum: A secure decentralized generalized transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] J. Liu, X. Sun and K. Song, "A food traceability framework based on permissioned blockchain," *Journal of Cyber Security*, vol. 2, no. 2, pp. 107–113, 2020.

[5] Q. Wang, F. Zhu, S. Ji and Y. Ren, "Secure provenance of electronic records based on blockchain," *Computers, Materials & Continua*, vol. 65, no. 2, pp. 1753–1769, 2020.

[6] X. Jiang, M. Liu, C. Yang, Y. Liu and R. Wang, "A Blockchain-based authentication protocol for WLAN mesh security access," *Computers, Materials & Continua*, vol. 58, no. 1, pp. 45–59, 2019.

[7] Z. T. Li, J. W. Kang, R. Yu, D. D. Ye, Q. Y. Deng *et al.*, "Consortium blockchain for secure energy trading in industrial internet of things," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 14, pp. 3690–3700, 2018.

[8] L. Luu, D. Chu, H. Olickel, P. Saxena and A. Hobor, "Making smart contracts smarter," in *Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security*, Vienna, Austria, pp. 254–269, 2016.

[9] C. F. Torres and J. Schütte, "Osiris: hunting for integer bugs in ethereum smart contracts," in *Proc. of the 34th Annual Computer Security Applications Conf.*, San Juan, Puerto Rico, USA, pp. 664–676, 2018.

[10] S. Kalra, S. Goel, M. Dhawan and S. Sharmar, "Zeus: analyzing safety of smart contracts," in *Proc. of the 25th Annual Network and Distributed Systems Security Symp.*, San Diego, California, USA, pp. 1–12, 2018.

[11] T. Chen, Z. Li and Y. Zhu, "Understanding ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.

[12] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier *et al.*, "Formal verification of smart contracts: short paper," in *Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, New York, NY, USA, 2016.

[13] M. Rodler, W. Li, G. O. Karame and L. Davi, "Sereum: protecting existing smart contracts against re-entrancy attacks," in *Proc. 26th Network & Distributed System Security Symp.*, San Diego, CA, USA, 2019.

[14] T. Chen, R. Cao, T. Li, X. P. Luo, G. F. Gu *et al.*, "SODA: A generic online detection framework for smart contracts," in *Proc. of the NDSS*, San Diego, CA, USA, 2020.

[15] B. Jiang, Y. Liu and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. of the 33rd ACM/IEEE Int. Conf. on Automated Software Engineering*, Montpellier, France, pp. 259–269, 2018.

[16] J. X. He, M. Balunovic, N. Ambroladze, P. Tsankov and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security (CCS '19)*, New York, NY, USA, pp. 531–548, 2019.

[17] Z. T. L. I., W. L. L. I., F. Y. Lin, Y. Sun, M. Yang *et al.*, "Hybrid malware detection approach with feedback-directed machine learning," *SCIENCE CHINA Information Sciences*, vol. 63, no. 139103, pp. 1–139103, 2020.

[18] Z. Y. Wei, J. Q. Wang, X. Q. Shen and Q. Luo, "Smart contract fuzzing based on taint analysis and genetic algorithms," *Journal of Information Hiding and Privacy Protection*, vol. 2, no. 1, pp. 35–45, 2020.

[19] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen *et al.*, "Reguard: finding reentrancy bugs in smart contracts," in *IEEE/ACM 40th Int. Conf. on Software Engineering: Companion (ICSE-Companion)*, Gothenburg, Sweden, pp. 65–68, 2018.

[20] P. Tsankov, A. M. Dan, D. D. Cohen, A. Gervais, F. Buenzli *et al.*, "Securify: practical security analysis of smart contracts," in *Proc. of the 2018 ACM Conf. on Computer and Communications Security*, Toronto, Canada, 2018.

[21] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *Proc. of the 27th USENIX Security Symp.*, Baltimore, MD, USA, pp. 1317–1333, 2018.

[22] L. Alt and C. Reitwiessner, "Smt-based verification of solidity smart contracts," in *Int. Symp. on Leveraging Applications of Formal Methods*, Limassol, Cyprus, pp. 376–388, 2018.

[23] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *Financial Cryptography and Data Security*, vol. 10323, pp. 520–535, 2017.

[24] I. Grishchenko, M. Maffei and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of Security and Trust*, vol. 10804, pp. 243–269, 2018.

[25] S. K. Lahiri, S. Chen, Y. Wang and I. Dillig, "Formal specification and verification of smart contracts for azure blockchain," in *Verified Software. Theories, Tools, and Experiments (LNCS)*, vol. 12031, pp. 87–106, 2019.