Grenoble INP **UGA**
Master CybserSecurity **Year 2021-22**

Software Security & Secure Programming (L. Mounier)

Written Exam - Tuesday January the 18th, 2022

Duration: 2 hours – Answers can be written either in English or in French.
All documents allowed apart books – Electronic devices are forbidden.

This exam contains two distinct parts:

1. One exercise, supposed to be solved in about 50 minutes;
2. Some questions on a research paper, allow 1h to read the paper and answer these questions.

Exercise. (∼ 10 pts)

We consider the following C program (on the left) and its Control Flow Graph (CFG, on the right):

```
1  #define N 7
2  #define L 15
3
4  int main () {
5    int x, y;
6    int T[L] ;
7    x = 1 ;
8    y = 0 ;
9    while (x<N) {
10       y = x+3 ;
11       x = x+2 ;
12   } ;
13   T[x+y] = 42 ;
14   return 0 ;
15 }
```

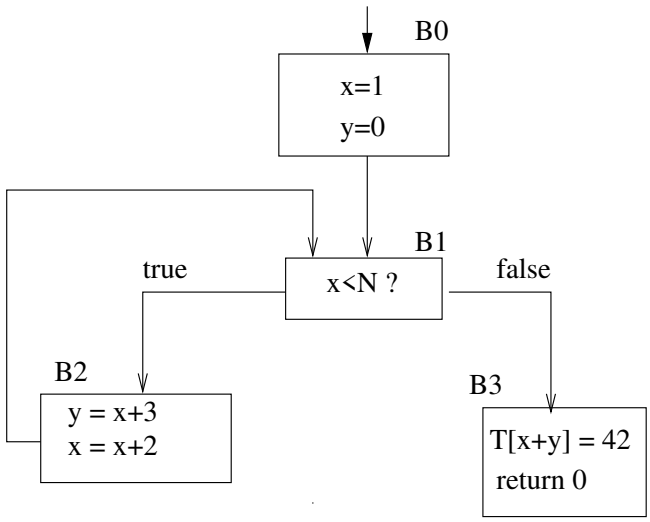


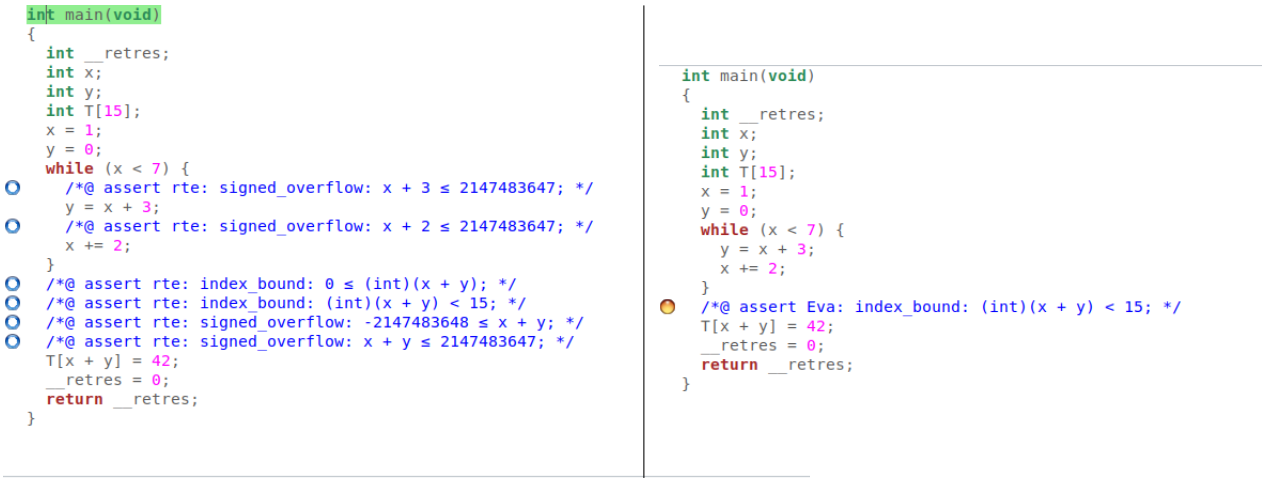
Figure 1: a C program and its Control-Flow Graph (CFG)

Q1. Figure 2 (left) is a screen copy of the results produced when running the `-rte` (runtime-error) option of Frama-C. Explain these results.

Q2. Figure 2 (right) is a screen copy of the results produced when running the `-eva` (value-set analysis) option of Frama-C. According to Frama-C what is (potentially) wrong with this program ?

Q3. We would like to know if the result produced by the value-set analysis of Frama-C is a *false positive*.

1. Compute (manually) a value-set analysis using **intervals** and give the abstract values obtained for variables `x` and `y` at the entry and exit locations of each basic block (as Frama-C did when running its value-set analysis). You can use widening/narrowing operators if you want (but it is not mandatory on this example). According to this computation, is your conclusion similar to the one produced by Frama-C ?



```
int main(void)
{
  int __retres;
  int x;
  int y;
  int T[15];
  x = 1;
  y = 0;
  while (x < 7) {
    /*@ assert rte: signed_overflow: x + 3 ≤ 2147483647; */
    y = x + 3;
    /*@ assert rte: signed_overflow: x + 2 ≤ 2147483647; */
    x += 2;
  }
  /*@ assert rte: index_bound: 0 ≤ (int)(x + y); */
  /*@ assert rte: index_bound: (int)(x + y) < 15; */
  /*@ assert rte: signed_overflow: -2147483648 ≤ x + y; */
  /*@ assert rte: signed_overflow: x + y ≤ 2147483647; */
  T[x + y] = 42;
  __retres = 0;
  return __retres;
}
```

```
int main(void)
{
  int __retres;
  int x;
  int y;
  int T[15];
  x = 1;
  y = 0;
  while (x < 7) {
    y = x + 3;
    x += 2;
  }
  /*@ assert Eva: index_bound: (int)(x + y) < 15; */
  T[x + y] = 42;
  __retres = 0;
  return __retres;
}
```

Figure 2: Results obtained with Frama-C when running `-rte` (left) and `-eva` (right)

2. According to your own understanding of this program behavior at runtime, is it a false positive ?

Q4. We now define the constant `L` with the value 50 and we replace the assignment of line 12 by `y = y+3`. What would be the abstract values obtained for variable `y` at each entry and exit locations of each basic blocks ? According to this computation what can we conclude regarding the correctness of this new code ?

Q5. We would like to check, using symbolic execution[1], if the program below may contain a buffer overflow at line 11, assuming that the `read()` function return any value of type `int`.

```
1  #define N 7
2  #define L 15
3
4  int main () {
5    int x, y;
6    int T[L] ;
7    y=read() ;
8    for (x=1 ; x<N ; x=x+2) {
9        y = x+3 ;
10   } ;
11   T[x+y] = 42 ;
12   return 0 ;
13 }
```

1. Give the path predicate (on variables `x` and `y`) leading to line 11. How do you ensure that the "for" loop is unrolled enough time ?
2. Give a solution (i.e., an input value of `y`) for this path predicate (making this path **executable**).
3. Extend this path predicate in order to get a formula allowing to trigger the (potential) buffer overflow vulnerability of line 11. Is there a solution (i.e., an input value of `y`) which satisfies this formula ?

[1] with a tool like PathCrawler

Q1

Frama-C is detecting potential errors which are:

① Overflow of variable y if $x+3 \geq MAX\_INT$

② Overflow of variable x if $x+2 \geq MAX\_INT$

③+④ Out of bound index if $x+y < 0$ or $x+y \geq 15$.

⑤+⑥ Overflow due to the sum operation if $x+y < MIN\_INT$ or $x+y > MAX\_INT$

Q2

Frama-C was able to discard all the above errors except for error ④

For the condition $x+y < 15$ he is not able to guarantee that an error will not occur

Q3

Performing the value set analysis manually:

$x = und$ $y = und$

Exiting B0: $x = 1$ $y = 0$

Exiting B1: $x = [7, +\infty]$, $y = [8, +\infty] \Rightarrow$ always $x = 7$, $y = 8$

Computing B3: $x + y = [15, +\infty] \Rightarrow$ always $x + y = 15$ Overflow!

Our conclusion is in agreement with frama-C, a buffer overflow will occur This means that it is not a false positive!

Q4

Exiting B0 $y = 0$

Exiting B1 $y = [9, +\infty] \Rightarrow$ always 9

Exiting B2 $y = [3, +\infty] \Rightarrow y = [3, 9]$

Computing B3 $x + y = 16$. no overlow will occur

Questions on a research paper. (∼ 10 pts)

Read **sections I to IV** of the paper given in appendix, answering the following questions as long as you read it. The objective of this part of the exam is to evaluate your ability to understand a description of a vulnerability detection tool (its strengths and limitations, with respect to other approaches you know). When answering the following questions you should not copy entire sentences from the paper but rather illustrate your point with examples and comments from your own.

Q1 [section I].

1. Why Etherum smart contracts vulnerabilities are a serious problem ?
2. How can we explain that some existing techniques classified as "unexploitable" the so-called "Exposed Secret" vulnerabilities ?

Q2 [section II]. Explain the 3 smart-contract vulnerabilities described in section II.C. To which kind(s) of exploitation category (as defined in section II.B) could lead each of these vulnerabilities ?

Q3 [section III]. Explain the 2 main challenges for smart-contract exploit generation and the approach used by EthPloit to address them.

Q4 [section IV (before section IV-A)].

1. Explain the main differences/specificity between EthPloit and a more generic fuzzer like AFL (i.e., which steps are added/removed in EThPloit with respect to AFL ?)
2. What are the output of EthPloit ? Is it always terminating ? *Successfully* terminating ?

Q2 [section IV-A].

1. Explain the notions of *taint sources* and *taint sinks*. Why taint analysis is used for in this context ?
2. We consider the following statement: `v1 = v2 + 1 - v2`
   Would `v1` be flagged as *variable data-dependent* from `v2` ? What would be the possible consequences (within EthPloit) ?
3. Explain the notion of *variable control-dependency*, giving an example of your own.
4. What is the output of the taint propagation step ?

Q3 [section IV-B].

1. (Taint Relation Graph): Give a small example of a taint relation graph on a (pseudo)-C code of your own
2. (Function Selection): Is it possible that EthPloit generate an empty taint relation graph ? If yes, what would be the consequences ? what is the objective of distribution P (you could better answer this question latter, after reading section IV-D1)
3. (Function Argument Generation): What is the expected gain when mixing two argument generation strategy (instead of using a single one) ? How does the tool "switch" from one strategy to another ?
4. (Blockchain Property Generation):

   > "The sender is selected from a predefined set of accounts, which represent the accounts owned by attacker"

   Is it explained how this set of accounts is chosen ? Is it important in practice ?

Q4 [section IV-C].

1. Why using an open-source debugger to implement the EVM environment ? What are the potential drawbacks with this solution ?
2. Why running some transaction twice ? What about false positive and false negatives ?

Q5 [section IV-D].

1. What are the main objectives of trace analysis ? Is this step also present in a fuzzer like AFL ??
2. Explain the so-called code injection oracle

Q6 [section IV-E]. Explain (with your own words) the last paragraph of this section, telling how dynamic seeds help to solve the Unsolvable Constraints problem. Is this solution always successful ? Explain your answer ...

Q7. According to you, what are the main strengths and weaknesses of the proposed approach ?

ETHPLOIT: From Fuzzing to Efficient Exploit Generation against Smart Contracts

Qingzhao Zhang*‡, Yizhuo Wang*‡, Juanru Li*, Siqi Ma†(✉)
*Shanghai Jiao Tong University, China
{fszqz001, mr.wang-yz}@sjtu.edu.cn, roman@sjtusec.com
† Data 61, CSIRO, Australia
siqi.ma@csiro.au

***Abstract*—Smart contracts, programs running on blockchain systems, leverage diverse decentralized applications (DApps). Unfortunately, well-known smart contract platforms, *Ethereum* for example, face serious security problems. Exploits to contracts may cause enormous financial losses, which emphasize the importance of smart contract testing. However, current exploit generation tools have difficulty to solve hard constraints in execution paths and cannot simulate the blockchain behaviors very well. These problems cause a loss of coverage and accuracy of exploit generation.**

To overcome the problems, we design and implement ETHPLOIT, a smart contract exploit generator based on fuzzing. ETHPLOIT adopts static taint analysis to generate exploit-targeted transaction sequences, a dynamic seed strategy to pass hard constraints and an instrumented Ethereum Virtual Machine to simulate blockchain behaviors. We evaluate ETHPLOIT on 45,308 smart contracts and discovered 554 exploitable contracts. ETHPLOIT automatically generated 644 exploits without any false positive and 306 of them cannot be generated by previous exploit generation tools.

***Index Terms*—smart contract, fuzzing, exploitation**

I. INTRODUCTION

Blockchain-based cryptocurrency systems gain great popularity in the past several years, standing at an overall market capitalization of 170 billion dollars in April 2019 [1]. *Ethereum* [2], for example, is the second-largest blockchain system and cryptocurrency after *Bitcoin* [3] by overall market value [1]. *Ethereum* develops Bitcoin's scripts, a piece of stack-based code performing some simple checks before currency transfer, to a Turing-complete on-chain programming language called smart contract. As a result, more than cryptocurrency, *Ethereum* serves as a platform for decentralized applications (i.e., DApps) based on smart contracts, such as tokens, gambling, auction, etc. The website *State of the DApps* [4] records over 2,200 active DApps, over 85% of which come from *Ethereum*. In fact, *Ethereum* had hosted over one million smart contracts by the end of 2018 [5].

As the smart contract develops, security vulnerabilities of *Ethereum* smart contracts become a serious problem. The Turing-complete language makes smart contracts more error-prone than Bitcoin's scripts and several vulnerabilities have been discovered [6]. Because of the nature of cryptocurrency, smart contracts usually involve flows of *ETH*, the virtual currency in *Ethereum* which is also valuable in the real world. Therefore, when attackers make use of vulnerabilities, the currency is illegally manipulated. In fact, real-world attacks such as DAO [7] and Parity Multisig Wallet [8] have caused a tremendous crisis for Ethereum. What is worse, smart contracts are unmodifiable after on-chain deployment and the damage of attacks is almost irretrievable.

One major research goal for smart contracts vulnerability is how to fulfill an accurate vulnerability detection. Existing detection techniques [9]–[12] often suffer from both false negative and false positive. To verify a detected vulnerability, a typical method is to generate an exploit to test whether the vulnerability can be actually triggered. Current exploit generation for smart contracts generally applies symbolic execution method [13], [14]. *Teether* [13], for instance, locates potential dangerous operations and solves an execution path triggering the operation using SMT solvers. However, such a workflow of symbolic execution faces two challenges. The first one is *Unsolvable Constraints*, which indicates the execution of sensitive operations (e.g., currency transfer, self-destruction) is restricted by conditions that are difficult for existing tools to pass. *Unsolvable Constraints* is a major obstacle for symbolic execution. A prominent case is the validity check of hash value before currency transfer. For instance, *teether* tries to jump over a hash instruction but cannot present value relation between hash pre-image and hash value, which does not help to pass cryptographic checks in many scenarios. Our observation on 49,522 contracts collected from *etherscan* [5] demonstrates that these complicated conditions are common as 20% (9,694) contracts contain cryptographic functions. In this situation, the corresponding exploit is not able to be generated. Another challenge for current techniques is how to handle *Blockchain Effects*. Blockchain properties (e.g., timestamp and block number) are variables used in the contracts, which represent information about objects (e.g., block) in the blockchain system. Current exploit generation tools regard blockchain properties as normal global variables. However, these properties have their meaning in the blockchain system and their value has a specific range. If not handled properly, it is also infeasible to generate an exploit successfully.

To respond to the above challenges, in this paper, we utilize fuzzing to generate exploits. Compared with symbolic execution, fuzzing does not need to mathematically solve

‡ These authors equally contribute to the paper.

execution paths but generates input candidates to scan the paths, therefore bypasses the dilemma of symbolic execution. In addition, it is easier for a fuzzing framework to simulate blockchain behaviors with runtime instrumentation. We implement ETHPLOIT, a smart contract exploit generator, which basically generates transaction sequences to test whether the subject contract is exploitable. To address the problem of *Unsolvable Constraints*, ETHPLOIT adopts a dynamic seed strategy to make use of runtime values as feedback, so that finds the solution of cryptographic functions from execution histories. To simulate *Blockchain Effects*, ETHPLOIT leverages an instrumented *Ethereum* Virtual Machine (EVM) to provide custom configurations, such as setting timestamps and reverting external calls. In addition, to minimize search space and make the fuzzing process more efficient, ETHPLOIT deploys a taint analysis to guide transaction sequence generation and rules out invaluable test candidates in the first place.

We evaluate ETHPLOIT with 45,308 on-chain *Ethereum* contracts, and discovered 554 contracts can be exploited successfully. ETHPLOIT automatically generated 644 exploits for those contracts without any false positive and toke less than two seconds to generate each exploit on average.

In comparison, existing exploit generation tools (*Teether* and *MAIAN*) only generate a subset (334) of exploits. Specifically, ETHPLOIT generates 112 exploits against *Exposed Secret*, a class of vulnerability involving hash functions and thus the existing tools label them as "unexploitable". The results demonstrated that ETHPLOIT significantly improved the exploit generation technique against smart contracts and developers could leverage our tool to build more secure code.

In summary, we make the following contributions:

1) We summarized two major challenges, *Unsolvable Constraints* and *Blockchain Effects*, in smart contract exploit generation, and proposed a corresponding solution (i.e., a fuzzing approach) to address them.
2) We design and implement ETHPLOIT to fulfill our fuzzing guided exploit generation. ETHPLOIT makes use of three key techniques (taint constraints, EVM instrumentation, and dynamic seed strategy) and is able to achieve a more effective exploit generation.
3) By utilizing ETHPLOIT, we discovered 544 vulnerable contracts from *Ethereum* blockchain and generated 644 valid exploits for them, among which 306 exploits are not discovered by previous tools.

II. SMART CONTRACT SECURITY ISSUES

In this section, we first define some important concepts used in this paper. Then, we introduce the existing smart contract exploits and the corresponding discovered vulnerabilities.

A. Definitions

***Definition** 1:* **Solidity variables**. Solidity defines four types of variables according to their purposes:

- *Local variable*. The variable declared in a function that is destructed when the function returns.
- *State variable*. The variable declared globally that indicates the state of the contract.
- *Function argument*. The variable declared as an input of a function that is provided by the sender of the transaction. Such variables are usually untrustworthy.
- *Blockchain property*. The variable representing properties of blockchain system that contains three types of variables, message properties (e.g., `msg.sender`), transaction properties (e.g., `tx.origin`), and block properties (e.g., `block.timestamp`) [2].

***Definition** 2:* **Transaction** refers to an *Ethereum* transaction invoking the execution of a smart contract. Each transaction consists of four elements: a *sender*, a *receiver* (i.e. the address of the target contract), a *value* (i.e., amount of currency), and a *data section* containing a called function with its corresponding arguments.

***Definition** 3:* **Test case** refers to a sequence of transactions. Each transaction in the test case is executed sequentially in positive order. If the test case identifies a vulnerability, itself represents a valid exploit.

B. Exploitation of Smart Contract

Vulnerabilities of smart contract platforms could happen at the blockchain level, EVM level, and contract level [12]. We focus the contract-level vulnerabilities. Through smart contract vulnerabilities, attackers are able to exploit them gaining control of assets or causing damages. According to the cause of damages, we classify smart contract exploits into three categories [13], [14]:

1) *Balance Increment.* Contracts send currency to arbitrary accounts. Attackers conduct value transfer to gain profits from contracts by controlling target contracts.
2) *Self-destruction. Ethereum* contracts implement a special operation of destroying themselves. Such a dangerous operation cannot be accessible by attackers.
3) *Code Injection.* There are operations importing codes from external contracts. Then attackers are allowed to inject arbitrary malicious code into the execution if they have the control of such external contracts.

We observe that the three categories of exploits usually cause an external call: currency transfer, self-destruction, and execution of external code, respectively. To trigger a successful exploit using a test case, two requirements should be followed: 1) a critical transaction, whose execution exploit contract vulnerabilities, must have at least one execution path to trigger one of the vulnerable external function calls; 2) a set of transactions must be executed to modify states of the contract before the critical transaction does.

C. Smart Contract Vulnerabilities

We introduce three types of vulnerability, where two types, *Unchecked Transfer Value*, and *Vulnerable Access Control*, are discovered by previous smart contract exploit generation tools [13] [14]. Another type, *Exposed Secret*, is a newly identified vulnerability for which previous smart contract exploit generation tools cannot generate valid exploits.

```
1  contract NewSmartPyramid {
2      function withdraw() notOnPause public {
3          if (block.timestamp >= x.c(msg.sender
               ) + 10 minutes) {
4              uint _payout = (x.d(msg.sender).
                   mul(x.getInterest(msg.sender))
                   .div(10000)).mul(
                   block.timestamp.sub(x.c(msg.
                   sender))).div(1 days);
5              x.updateCheckpoint(msg.sender);
6          }
7          if (_payout > 0)
8              msg.sender.transfer(_payout);
9      }
10 }
```

Listing 1: A contract with block timestamp for investment.

```
1  contract HOTTO is ERC20 {
2      owner = msg.sender;
3      function HT() public {
4          owner = msg.sender;
5          distr(owner, totalDistributed);
6      }
7      function withdraw() onlyOwner public {
8          address myAddress = this;
9          uint256 etherBalance = myAddress.
               balance;
10         owner.transfer(etherBalance);
11     }
12 }
```

Listing 2: A contract with *Bad Access Control* vulnerability

1) Unchecked Transfer Value: It describes the amount of currency transfer without being constraint including two types of implementations.

First, misuse of `this.balance`. It indicates that the total transfer balance is not being checked by a contract, which causes economic losses.

Second, unlimited profit. A large number of smart contracts provide users with high-yield investments in the form of tokens or games. It helps attackers to gain unlimited profits if they manipulate the calculation of rewards. For example in Listing 1, since `_payout` is dependent on `block.timestamp` (Line 5) and it is allowed to be transferred without any constraints (Line 10), an attacker can gain any profits as long as the contract balance is not exceeded.

2) Vulnerable Access Control: It represents that an attacker can bypass access control or grant privileges to conduct sensitive operations, such as currency transfer and self-destruction. Consider Listing 2 as an example, by executing public function `HT`, which changes original `owner` to `msg.sender` with no checks (Line 4), an attacker can bypass `onlyOwner` checking (Line 7) to withdraw all currency in contract.

3) Exposed Secret: It represents that an attacker can provide a secret value, which should be private to attackers, to access sensitive operations. Contracts with this vulnerability usually work as an application of gambling, quiz, gift and so on. Typical contracts of this type include two main functions: One is *Secret Setter*. The function sets the secret value and store the secret (or some transformation of the secret) in the contract state variables. The other one is *Secret Checker* which

```
1  contract Game {
2      address questionSender;
3      string public question;
4      bytes32 responseHash;
5      function Try(string _response) external
           payable{
6          require(msg.sender == tx.origin);
7          if(responseHash == keccak256(_response)
               && msg.value>1 ether){
8              msg.sender.transfer(this.balance);
9          }
10     }
11     function StartGame(string _question,string
           _response) public payable {
12         if(responseHash==0x0){
13             responseHash = keccak256(_response);
14             question = _question;
15             questionSender = msg.sender;
16         }
17     }
18 }
```

Fig. 1: A gaming contract with hash function.

accepts some proofs to verifies whether the sender knows the secret and then execute sensitive operations such as currency transfer or writing state variables.

Unluckily, because of the openness of blockchain, attackers can inspect the secret by observing the data of previous transactions and then break the secret checking. The *Game* contract in Figure 1 is a typical example. The secret setter `StartGame` stores the hash value of the secret value, which is a common approach to handle passwords. However, the plain text of the secret *_response* is a transaction argument, which is recorded publicly in the blockchain. Formally, if any secret value appears in the contract execution as plain text, the secret value can be conducted from past transactions and we regard this contract is vulnerable for *Exposed Secret*.

III. Motivation

In this section, we list challenges to exploit smart contract vulnerabilities and propose our approaches to address the corresponding challenges.

A. Challenges of Smart Contract Exploit Generation

By analyzing previous smart contract exploit generation tools [13] [14], we summarized two challenges for exploit generation, *Unsolvable Constraints* and *Blockchain Effects*.

1) Unsolvable Constraints: Sensitive operations in smart contracts are commonly restricted by conditions such as validity check of a hash value. Thus, solving such constraints is necessary for generating exploits. Previous tools (e.g., *Teether*, *Mythril*) rely on SMT solvers to address path constraints. However, SMT solvers cannot solve the constraint if it involves complicated operation like hash (opcode `SHA3`). Although some improvements are made in *Teether* (e.g., iterative constraint solving), path constraints cannot be solved completely.

Figure 1 demonstrates an example with a cryptographic constraint (Line 7). Operation `transfer` (Line 8) is triggered if the value of `responseHash` equals to the return value of the hash function `keccak256`. Variable `responseHash` is assigned by hash function `keccak256`

in function `StartGame` (Line 13), which is unable to be traced by previous tools.

2) Blockchain Effects: Blockchain effects of blockchain system such as blockchain properties affect the execution of smart contracts. An attacker may control the blockchain effect and execute sensitive operations (e.g., `transfer`) if they are dependent on the blockchain effect. To exploit such vulnerability, the blockchain effect is required to be implemented reasonably. For instance, the timestamp must be represented as the current time or recent time, instead of an exaggerated time. For example, the contract shown in Listing 1 demonstrates a transfer correlated to a blockchain effect, i.e., `block.timestamp`. Variable `_payout` is the amount of transfer (Line 10) and it is assigned by a calculation, which includes `block.timestamp` (Line 5). If selecting a proper timestamp, attackers can gain profit from the contract. The current exploit generation tool, *Teether*, generates an invalid timestamp to exploit this vulnerability without considering the syntax of the `block.timestamp`.

B. Approaches for Challenges

Fuzzing [15] is an automated software testing technique which is commonly used to generate exploits for security-critical programs. In order to fuzz smart contracts for exploit generation, we apply a smart contract specific fuzzer which targets on exploitation. To address the above challenges, we leverage the following approaches to fuzzer:

Feedback of runtime values. We record the runtime values of arguments and variables to solve the defect of *Unsolvable Constraints*. Initially, we create a blank seed set for each argument of each function. According to the execution of previous transactions, we update the seed set of each function argument by adding runtime values, including previous function arguments, state variables and so on. We then apply the seeds as the feedback to generate arguments for the next transaction to execute. In this way, the feedback indicates the execution history and current state of the contract, which helps to generate valuable exploits. With the hash input recorded in the seed set, we can pass the validity check of the hash value, which is the most challenging case of *Unsolvable Constraints*.

Manipulation of blockchain execution. We propose to instrument the execution environment of smart contracts to support configurable executions, which means we can freely set the value of blockchain properties to solve *Blockchain Effects*. For example, we can configure each transaction execution by setting specific block timestamp which is realistic while able to exploit vulnerabilities, therefore explore more possibilities of contract execution.

IV. ETHPLOIT: SMART CONTRACT FUZZER

We design ETHPLOIT, a fuzzer for exploiting smart contracts automatically. Figure 2 depicts the workflow of ETHPLOIT, which consists of five steps:

1) **Static Analysis.** Given Solidity code of smart contracts, ETHPLOIT compiles the code to extract the Application Binary Interface (ABI) and the bytecode. It also applies taint analysis to extract dependencies among variables.
2) **Test Case Generation.** ETHPLOIT initially generates a test case. Instead of providing any predefined code patterns, ETHPLOIT conducts fuzzing to optimize the test case. The test case is further updated based on the static analysis results and feedback results generated by the previous round of fuzz test.
3) **Test Case Execution.** ETHPLOIT instruments an EVM to simulate blockchain effects. It further applies each test case on the instrumented EVM and output execution traces. The traces cover more execution probabilities because of the instrumentation.
4) **Trace Analysis.** ETHPLOIT analyzes each execution trace. If the exploit detector identifies an exploit, ETHPLOIT reports the current test case as a valid exploit. Meanwhile, coverage guider constructs feedback for optimizing the test case. ETHPLOIT distinguishes the feedback into two categories: function reward for modifying function distribution and valuable seeds for updating dynamic seed sets.
5) **Feedback Handling.** ETHPLOIT regards the function reward as function distribution for test case generator to select functions that are more likely to exploit vulnerabilities by the experience of past fuzzing iteration. Feedback handler also adds valuable seeds into seed sets which can be used to generate arguments able to address some strict constraints.

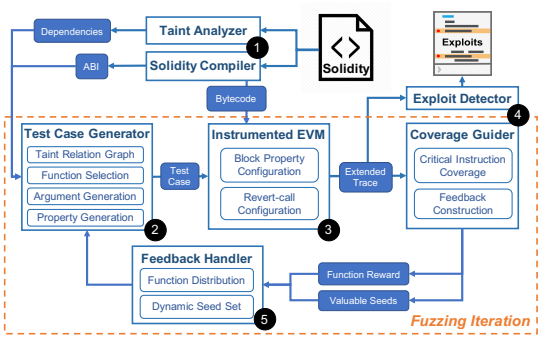


Fig. 2: Workflow of ETHPLOIT: (1) Static analysis; (2) Test case generation; (3) Test case execution; (4) Trace analysis; (5) Feedback handling.

ETHPLOIT executes Step (2) - Step (5) repeatedly until the total amount of the generated test cases exceeds a threshold.

To implement the approaches stated in Section III-B, ETHPLOIT proposes three key techniques:

Dynamic Seed Strategy. The feedback of runtime values is the seeds in ETHPLOIT, which guides argument generation for each transaction. Under a dynamic seed scheduling assisted by taint analysis, seeds are precisely fed into specific arguments.

EVM Instrumentation. To provide light-weight blockchain manipulation, ETHPLOIT deploys an instrumented EVM envi-

ronment that works the same as the official EVM and leverages instrumentation to simulate blockchain effects.

Taint Constraints of Test Case Generation. To minimize the search space of fuzzing, we only generate valuable test cases based on taint analysis. ETHPLOIT makes sure that the functions of each test case have internal dependencies.

A. Static Analysis

Taking the source code of each contract (i.e., Solidity program) as input, ETHPLOIT first compiles the source code and applies taint analysis to learn the dependencies of variables from the source code.

1) Solidity Compiler: ETHPLOIT uses the solidity compiler, *solc* [16], to extract its bytecode and ABI. The bytecode is used for deploying the contract in test case execution. ABI is taken as the input of test case generation.

2) Taint Analyzer: As each processed transaction may modify states of the contract, ETHPLOIT applies static taint analysis to discover dependencies of modifying contract states.

Our taint analyzer is built on top of *Slither* [17], which is an open-source static analyzer for parsing the smart contract source code. The taint analyzer proceeds the following steps to extract dependencies in a smart contract.

Control Flow Graph Generation. The taint analyzer first creates a Control Flow Graph (CFG) for each function, where each node is a Solidity expression (e.g., assignments, function calls, branch operations) and each edge connecting two nodes that are executed sequentially. From each CFG node, we can extract a set of read variables and a set of written variables. First, the analyzer creates an `ENTRY` node and an `EXIT` node to represent the beginning and the end points, respectively. It then parses the function and extracts conditional branches such as `IF` and `FOR`. For each conditional branch, the analyzer follows its execution sequence of expressions to construct a graph. Analyzer finally connects all graphs to generate a CFG.

Taint Source and Sink Labeling. Given the CFG of a function, analyzer labels taint sources and sinks by analyzing the expression of each node.

The analyzer labels state variables, function arguments, and blockchain properties as taint sources. These variables are a potential risk, as values of these variables are directly/indirectly assigned by the sender of the transaction. It then marks state variables and external calls (i.e., *send*, *transfer*, *call*, *callcode*, *delegatecall*, *selfdestruct*) as taint sinks. Due to the contract states stored in state variables are not destructed when the function returns, it is essential to continue tracking the states through state variables. Besides, external calls are the only trigger to explore smart contract exploits. These taint sources and sinks are used for further taint propagation.

Taint Propagation. According to the labeled taint sources and sinks, the taint analyzer learns how tainted value propagates in the function. Hence, the analyzer proceeds taint propagation by analyzing the dependencies of variables. Since we only consider the tainted value propagation, we extract variable-level dependencies, which are defined below:

1) *Variable-Data Dependency*. A variable v_1 is variable-data dependent on a variable v_2 if the value of v_1 is assigned by v_2 such as $v_1 = v_2 + 1$.
2) *Variable-Control Dependency*. A variable v_1 is variable-control dependent on a variable v_2 if the expression with v_1 is control dependent on the expression with v_2. Consider code $if(v_2 != 0)\{v_1 = v_1+1;\}$ as an example, v_1 is variable-control dependent on v_2.

Starting from the `ENTRY` node in the CFG, the taint analyzer traverses the CFG iteratively and propagates taint on nodes. The propagation takes three steps on each CFG node:

1) *Initialize taint status*. If the CFG node has no predecessors, the analyzer initially creates a flow set for each taint source, represented as $Taint(src) \leftarrow \{src\}$. Otherwise, the analyzer creates $Taint$ of the current node by taking the union of predecessors' $Taint$. $Taint$ indicates the status of taint propagation at the current node.
2) *Extract immediate dependencies*. In the Solidity expression of the current node, written variables are variable-data dependent on read variables. Also, the analyzer locates CFG nodes that the current node is control dependent on. Written variables of the current node are variable-control dependent on read variables of these control dependent nodes.
3) *Update taint status*. For each extracted dependency in step 2, v_1 is variable-dependent on v_2 for example, the analyzer inserts v_1 to the flow set of sources that taints v_2, namely, for each src, $Taint(src) \leftarrow Taint(src) \cup \{v_1\}$ iff $v_2 \in Taint(src)$.

After proceeding propagation on each CFG node, We get the $Taint$ of the `EXIT` node, denoted by $Taint_{EXIT}$, as the taint analysis result of current function. A source src taints a sink $sink$ if $sink \in Taint_{EXIT}(src)$.

B. Test Case Generation

Test case generator takes the results of ABI and dependencies of variables as input, it then creates a test case to exploit smart contract vulnerabilities. In order to prevent the involvement of manual efforts, generator optimizes the test case through fuzzing, instead of manually defined patterns.

A test case is generated in the format of $I = (tx_1, tx_2, \ldots, tx_K)$, where tx_i is a transaction. We first introduce a representation of Taint Relation Graph depicting dependencies among variables in one test case. The generator then takes the following three steps to generate a test case: *Function Selection*, *Argument Generation*, and *Property Generation*. Details are introduced below.

1) Taint Relation Graph: The generator aims to optimize the test case by analyzing how attackers' inputs affect the execution of exploits. First, since the test case is a sequence of transactions, we need to learn dependencies among the function sequence more than in-function dependencies. Second, we only focus on dependencies that are relevant to exploitation. To achieve the above goals, for each test case, generator constructs a Taint Relation Graph that helps function selection (Section IV-B2) and dynamic seed feeding (Section IV-E).

Each node in the Taint Relation Graph is taint sources or taint sinks. A directed edge represents a dependency from one taint source to one taint sink. A state variable, which is globally used in one contract, can be one node though it is referred to in various functions. Rather than analyzing all variable dependencies in the test case, the generator only selects those that are related to external calls because external calls are the only way to trigger exploits. Given a test case and variables dependencies extracted from taint analysis, the generator traverses the transaction sequence from tx_K to tx_1 for constructing a Taint Relation Graph:

1) For tx_K, the generator finds all sources that taint external calls. It adds edges from sources to calls. Among these sources, the generator selects state variables and puts them into a set called *Target Sinks*.
2) From tx_{K-1} to tx_1, for each transaction, generator finds out all sources that taint any state variables in *Target Sinks*. The generator further adds edges from the sources to the sinks and updates *Target Sinks* by adding state variables from these sources.

Given Taint Relation Graph of the test case, if there is a path from a taint source to any external call in tx_K, we can learn that the source (indirectly) taints the calls and is valuable to fuzz for exploit discovery.

Taken the Figure 1 as an example, where solid arrows denote the edges and dotted lines connect the same variables, generator analyzes the test case (tx_1, tx_2) where tx_1 calls `StartGame` and tx_2 calls `Try`. Generator first identifies an external call `transfer` in function `Try`, which is variable-control dependent on a state variable `responseHash`. Generator then regards `responseHash` as a sink and further explores that it is directly variable-data dependent on a function argument `_response` in function `StartGame`. The Taint Relation Graph is updated as `_response` $\rightarrow$ `responseHash` $\rightarrow$ `transfer`, which indicates that `_response` in `StartGame` is valuable to fuzz.

2) Function Selection: To generate an executable test case, the first step is to select K functions from the contract for the K transactions, respectively.

Only non-static callable functions are suitable for transaction generation. First, the function must be public and callable, otherwise, it cannot be triggered by any transaction calls. Second, the function must be non-static, which contains external calls or operations to modify state variables, otherwise, it is useless for exploitation.

After removing improper functions, generator selects functions from tx_K to tx_1. For each transaction tx_i $(1 \leq i \leq K)$, we take the following two steps to select tx_i's function $tx_i.f$.

Candidate selection based on dependencies. While preparing to choose $tx_i.f$, generator takes the Taint Relation Graph of tx_K to tx_{i+1} and the corresponding *Target Sinks* as input. If the graph can be extended by one function, the generator adds the function to a set of candidates. More specifically, if $i = K$, the function is the one with at least one external function call. If $i < K$, the function is the one including inputs which indirectly taint the external calls in tx_{K-1}.

Function selection based on probability distribution. The generator then randomly selects a function from the candidates with a probability distribution. Basically, the distribution mappings functions to positive numbers, whose value is based on execution feedback. Given the distribution P and function candidates F, a function $f \in F$ has a probability of $P(f)/\sum_{i \in F} P(i)$ to be selected. Details of P is illustrated in Section IV-D1. Finally, $tx_i.f$ is the selected function.

3) Function Arguments Generation: After the function selection, the generator fills in arguments and block properties of each transaction to make it executable. In general, arguments are generated from two sources: pseudo-random generator and seeds. For the predefined probability p, it indicates that each argument has a probability p to be generated randomly. Besides, probability $1-p$ is to use the value of one seed from the corresponding seed set.

Random generation is based on types of arguments, which are divided into static-size (e.g., `uint256` and `bytes32`) and dynamic-size (e.g., arrays). For static-size arguments, the pseudo-random generator produces random values within the input domain. For example, range from 0 to $2^{256} - 1$ for `uint256` type. For dynamic-size arguments, since elements of them are static-size in Solidity version *0.4*, the generator first randomly selects a valid number as the length and then generates each static-size element.

On the other hand, the generator has a dynamic variable-specific seed strategy, which means that each argument in each function has a seed set. All seed sets are initialized with an empty set and new seeds are added into seed sets after the execution of transactions (details are in Section IV-E). By using the value from seeds, the generator makes use of the feedback of runtime values.

Therefore, to accept the feedback of runtime values, arguments for tx_i are generated when all executions of tx_j $(j < i)$ are done and dynamic seed sets are updated.

4) Blockchain Properties Generation: Blockchain properties that need to generate include message properties and block properties. There are two message properties to be generated: `msg.sender`, indicates the sender of the transaction, and `msg.value`, indicates the amount of currency the transaction carries with. The sender is selected from a predefined set of accounts, which represents the accounts owned by attackers. Note that the creator of the contract is apart from this account set. The generation of `msg.value` is the same as generation of arguments since it is also an `uint256` value. In addition, generator checks the function to be `payable` before generating value. Otherwise, the value should be zero.

`block.timestamp` and `block.number` indicate the timestamp and height of block which contains the current transaction, respectively. We instrument EVM environment to configure these block properties for each transaction execution. For example, the timestamp of a transaction can be the timestamp of the previous transaction plus a random period (e.g., 30 days in maximum).

After the above generation processes, a transaction is finally complete and ready to be executed.

C. Instrumented EVM Environment

EVM environment is used to deploy contracts and execute transactions. We implement ETHPLOIT's instrumented EVM environment based on *remix-debugger* [18], an open-source debugger for Solidity contracts. The environment can extract full execution trace of a transaction, including stack and memory status when processing each opcode. Compared with the private *Ethereum* chain used in many smart contract analysis tools, the debug environment is more fast and flexible to configure. To solve the problem of *Blockchain Effects* defined in Section III-A, we deploy three light-weight instrumentation.

First, instrumented EVM configures accounts to do initialization for each test case. Before each test case proceeds, instrumented EVM sets all accounts' balance to a default value and deploys the target contract. Each newly deployed contract is assigned by some initial balance.

Second, instrumented EVM configures block properties for each execution. The environment provides APIs for test case generator to generate block properties for each transaction during the block generation in the environment.

Third, the environment can force any external call to throw exceptions. In Ethereum, external calls, especially `send`, maybe failed accidentally because of the bad destination, lack of gas or insufficient balance. Such failure of calls can lead the execution to other paths which may contain error handling logic. However, many analysis tools (e.g., *Teether* and *ContractFuzzer*) ignore this possibility and lose the coverage of code. To improve the coverage of exploitation, transactions with external calls are executed twice. In the first execution, the transaction normally proceeds but in the second execution, the environment throws exceptions inside external calls.

D. Trace Analysis

Coverage guider and *exploit detector* both perform analysis on the contract execution trace extracted from the instrumented EVM environment. *Coverage guider* rewards the discovery of new execution paths and provides runtime information to the feedback handler. *Exploit detector* detects whether the execution triggers vulnerabilities and performs a successful exploit. If so, the exploit detector then reports the exploit.

1) Coverage Guider: To measure the progress of exploit-oriented fuzzing, we introduce a new coverage criterion based on statement coverage [19]: *critical instruction coverage*.

Rather than take all statements into consideration, critical instruction coverage only focuses on the critical instructions including: `SSTORE` which saves a value to contract state, and `SELFDESTRUCT`, `CALL`, `CALLCODE`, `DELEGATECALL` which make external calls. These instructions are indispensable for a successful exploit. If there are newly reached critical instructions in the execution trace, the coverage guider tags the corresponding test case as a coverage increment. An event of coverage increment triggers two effects as rewards.

First, *Seed feedback*. Some runtime values (e.g., arguments, state variables, etc) in the execution will be selected as new seeds, which guide variable generation in later rounds under dynamic seed strategy (Section IV-E).

Second, *function distribution feedback.* Recall that function selection is based on a probability distribution P. Since the current sequence I increase the coverage, we increase the probability of selecting functions of I in the future. In feedback handler, we use a simple method to calculate a probability distribution P. For each function f, $P(f) = c_0 + \frac{N_c}{N_t}(c_1 - c_0)$, where N_c denotes the number of executions of coverage increment, N_t denotes the total number of executions, c_0 and c_1 are the minimum and maximum boundary.

2) Exploit Detector: Exploit detector uses three oracles to detect the exploits defined in Section II-B.

The Balance Increment oracle detects whether attackers can gain financial profits by exploiting a contract, Recall that instrumented EVM sets a group of accounts as the attackers' accounts. After the execution of a test case, the oracle collects the balance of attackers' accounts and compares the current balance with the default initial one. The oracle claims a successful Balance Increment exploit if the balance of attackers' accounts increases.

The Self-destruction oracle detects whether the contract is unexpectedly destroyed. The oracle analyzes the execution traces of a test case and if it finds the opcode `SELFDESTRUCT` in the trace, it claims a successful exploit since the opcode `SELFDESTRUCT` is the only way to destroy a contract.

The Code Injection oracle detects whether attackers can inject codes into the execution of the contract. The oracle first finds opcodes `CALLCODE` and `DELEGATECALL`, which can import external codes. Second, the detector checks whether the destination of the two opcodes is controlled by the attacker. To be detailed, when EVM is going to execute the two opcodes, the detector accesses the program stack, fetch the destination value and check whether the destination is in the set of attackers' accounts. If there is the code-injection opcode whose destination is controlled, the oracle reports an exploit.

E. Dynamic Seed Strategy

As mentioned in Section IV-B, our seed strategy is part of feedback handling, which aims to guide the test case generator to produce proper function arguments. The seeds are selected mainly based on execution feedback from trace analysis.

Each argument in each function has a seed set that is initialized to an empty set. The strategy is dynamic because there are new seeds adding into seed sets after each execution of transactions and each execution of test cases. These new seeds indicate knowledge about previous executions and guide future argument generation.

There are two types of seeds in the strategy: global seeds and local seeds. Global seeds have a lifetime during fuzzing one contract, similar to the seeds in traditional coverage-guided fuzzing. When receiving a notification of coverage increment from the coverage guider, all arguments of that execution are added into the corresponding global seeds, which help searching deeper paths.

Local seeds are designed for transaction sequences especially. These seeds are initialized when starting to execute a

test case and are updated after every single transaction is executed and before generating the next transaction's arguments. In details, local seeds have the following sources:

1) *Previous arguments*. Once a transaction is executed, its arguments will be recorded as local seeds.
2) *State variables*. After a transaction execution, the coverage guider will collect the latest values of state variables and labels them as local seeds.
3) *I/O of complicated calls*. Cryptography functions, for example, are hard to predict or solve. We collect inputs and outputs of these complicated functions as seeds.
4) *Constant values*. For each function, we collect magic numbers or literal values in its function body as seeds.

Next, we solve the challenge of feeding seeds accurately to target arguments, mentioned in Section III-B. Suppose we have got a set of local seeds with various types before generating a transaction's arguments, we should filter the seeds for valuable ones and mapping the valuable seeds to corresponding target arguments. Seed feeding should satisfy two rules:

- *Type match*. The seed should have the same (or transformable) type as the target input argument.
- *Taint constraint*. The seed and target argument must taint the same taint sink. Formally, for a taint sink in Taint Relation Graph, there is a path from the seed to the sink and another path from the target argument to the sink.

As an example, we illustrate how dynamic seeds solve *Unsolvable Constraints* in contract *Game* (Figure 1). Suppose we generate a test case (`StartGame`, `Try`) (here we use function to represent transaction) and the target hash check is in `Try`. After the execution of `StartGame`, the value of `StartGame`'s argument `_response` is fed into `Try`'s argument `_response` as a local seed so that it is possible to pass the hash check. This seed feeding is qualified since two arguments are both in `string` type and both taints `transfer`. Also, the input of `keccak256` in `StartGame` is added into local seeds for `Try`'s argument `_response`. In a similar way, it can help pass the hash check.

V. Evaluation

In this section, we assess the performance of ETHPLOIT by conducting two experiments. First, we evaluate its effectiveness of generating exploits and compare the result with state-of-art tools *Teether* and *MAIAN*. Second, we assess the contribution of three key technologies to fuzzing efficiency.

A. Experiment

Dataset. We collected 49,522 smart contracts with verified source code from Etherscan [5] by the December of 2018. After removing the duplicated ones and those requiring deprecated compiler versions, we finally got 45,308 smart contracts for further experiments.

Setup. ETHPLOIT is written in 3,846 LOC *python* for fuzzing and 397 LOC *nodejs* for the instrumented EVM environment by modifying *remix-debugger* [18]. We use *solc* with the version of *0.4.25* as the solidity compiler. We ran our experiment on a server with Ubuntu 18.04, Intel(R) Xeon(R) Gold 5122 CPU @ 3.60GHz and 128GB RAM. We set the maximum fuzzing test cases for a smart contract as 1,000 (i.e., $T_{max} = 1,000$). Our results confirm that the 1,000 test cases are sufficient since most exploits are discovered in 100 test cases. It indicates that ETHPLOIT either generates an exploit after at most 1,000 fuzzing tests or regards the contract as secure. According to the results reported by *Teether* and *MAIAN*, most smart contract exploits can be found by transaction sequences whose length is at most three. Therefore, we set the maximum length for a test case as three ($K = 3$).

B. Evaluation of Smart Contract Exploits

Results of ETHPLOIT. We applied ETHPLOIT to the entire dataset and completed the experiment in 100 hours. In total, ETHPLOIT discovered 554 exploitable contracts. Since some contracts expose multiple exploitable vulnerabilities and We regard two exploits are different if the function of the last transaction ($tx_K.f$) is different. ETHPLOIT totally generated 644 exploits, which are verified using real-world EVM, including 600 Balance Increment, 59 Self-destruction, and 4 Code Injection. Also, we show details of 12 typical exploitable contracts in Table II.

The classification of exploits tells the consequences of exploits. To further learn the kind of vulnerability these exploits trigger, we manually inspected all generated exploits and observed that ETHPLOIT exploited all the vulnerabilities discussed in II-C. As shown in Table I, ETHPLOIT identified 112 *Exposed Secret*, 351 *Unchecked Transfer Value*, 142 *Vulnerable Access Control*, and 39 others, as Table I shows.

For *Exposed Secret*, 104 out of 112 exploits have cryptographic checks in the execution path. Corresponding contracts have a *secret setter* that accepts secret value as an argument, hashes it and stores the hash value to state variables. The *secret checker* then accepts hash value as input, hashes it and compares the hash output with the stored hash value. For these contracts, ETHPLOIT uses dynamic seeds to fetch secret values from previous execution and solves the constraint of *secret checker*, while symbolic execution cannot succeed in this. In the other eight exploits, the corresponding contracts directly save the plain text of the secret to state variables without hashing it. Even random fuzzing can exploit these contracts but the dynamic seeds make the exploit discovery faster.

Among *Unchecked Transfer Value*, ETHPLOIT triggers 144 *Unlimited Profit*, including 127 with high-yield investments, where the profit is calculated based on block number or timestamps. ETHPLOIT can simulate these block properties to gain profit from these contracts. The other 17 contracts are lottery games that use random numbers to calculate the value of currency transfer. Since they use block properties as random seeds, ETHPLOIT can exploit them by random fuzzing. Another 181 contracts have misused `this.balance`. We recommend using state variables to record expected transfer values with explicit limits rather than using `this.balance`.

Most *Vulnerable Access Control* comes from ERC20-Token contracts which have the same problem as the *HOTTO* (List-