Computer Modeling in
Engineering & Sciences

Tech Science Press

DOI: 10.32604/cmes.2021.015809

ARTICLE

An Adversarial Smart Contract Honeypot in Ethereum

Yu Han[1], **Tiantian Ji**[1], **Zhongru Wang**[1,2,*], **Hao Liu**[3,*], **Hai Jiang**[4], **Wendi Wang**[1] **and Xiang Cui**[5]

[1]Key Laboratory of Trustworthy Distributed Computing and Service (BUPT), Ministry of Education, Beijing University of Posts and Telecommunications, Beijing, 100876, China

[2]Chinese Academy of Cyberspace Studies, Beijing, 100010, China

[3]Qianxin Technology Group Co., Ltd., Beijing, 100088, China

[4]Beijing DigApis Technology Co., Ltd., Beijing, 100081, China

[5]Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou, 510006, China

[*]Corresponding Authors: Zhongru Wang. Email: wangzhongru@bupt.edu.cn; Hao Liu. Email: liuhao@qianxin.com

Received: 15 January 2021 Accepted: 11 March 2021

ABSTRACT

A smart contract honeypot is a special type of smart contract. This type of contract seems to have obvious vulnerabilities in contract design. If a user transfers a certain amount of funds to the contract, then the user can withdraw the funds in the contract. However, once users try to take advantage of this seemingly obvious vulnerability, they will fall into a real trap. Consequently, the user's investment in the contract cannot be retrieved. The honeypot induces other accounts to launch funds, which seriously threatens the security of property on the blockchain. Detection methods for honeypots are available. However, studying the manner by which to defend existing honeypots is insufficient to fight against honeypots. The new honeypots that may appear in the future from the perspective of an attacker must also be predicted. Therefore, we propose a type of adversarial honeypot. The code and behavioral features of honeypots are obtained through a comparative analysis of the 158,568 non-honeypots and 352 honeypots. To build an adversarial honeypot, we try to separately hide these features and make the honeypot bypass the existing detection technology. We construct 18 instances on the basis of the proposed adversarial honeypot and use an open-source honeypot detection tool to detect these instances. The experimental result shows that the proposed honeypot can bypass the detection tool with a 100% ratio. Therefore, this type of honeypot should be given attention, and defensive measures should be proposed as soon as possible.

KEYWORDS

Honeypot; smart contract; adversarial; bypass detection

1 Introduction

A smart contract is a computer program or a transaction protocol that is intended to automatically execute, control, or document legally relevant events and actions according to the terms of a contract or an agreement [1–3]. Ethereum is the first public blockchain to support smart contracts. Accounts established on Ethereum are divided into External Owned Account (EOA in short) and Contract Account (or Contract directly). EOAs are controlled by anyone with the private keys and can initiate transactions. Contract accounts are controlled by the smart contract code and can only send transactions in response to receiving a transaction.

A honeypot [4–7] is a special type of smart contract. The contract design exhibits maybe-obvious-flaws that if the user transfers a certain amount of funds to the contract, then the user can withdraw the funds in the contract. However, the user will fall into a real trap and cannot obtain back the funds invested in the contract once trying to take advantage of this defect. In this situation, the user who constructs this contract is called the attacker; the user who is deceived by this contract is called the victim. Honeypots use a phenomenon that the victim only pays attention to the maybe-obvious-flaws of the contract while ignoring the hidden crisis.

Hundreds of honeypots deployed on the Ethereum main chain have been discovered, and they can be divided into ten types according to various techniques. Honeypots induce the victim to actively send funds, threatening the security of property on the blockchain [8,9].

With the large-scale application of blockchain technology, smart contracts on the main chains are boosted. However, security issues have always been stressed due to the publicity of smart contracts [10,11]. Research on honeypots focuses on measurement and detection. Torres et al. [12] first defined the honeypot in "The Art of Scam". They also measured 150,000 smart contracts in the way of automated detection and manual analysis and finally obtained 282 honeypots. Besides, they proposed a tool combining symbolic execution and precise heuristic functions, namely, HONEYBADGER. The tool consisted of three components: symbolic, cash flow, and honeypot analytical components. Specifically, the EVM bytecode of the contract was taken as an input. The functions of the three components are as follows:

1) The symbolic execution component is used to construct the control flow graph and execute different paths.
2) The cash flow analytical component is used to determine whether the contract can accept and transfer funds on the basis of the result of symbolic execution.
3) The honeypot analytical component combines heuristic methods and symbolic execution to perform classification.

The experimental results demonstrated that the detection method based on HONEYBADGER was effective. Nonetheless, the detection method based on symbolic execution was limited to the detection of honeypots with known technology. It is not effective for certain honeypot types.

In contrast with bytecode-based detection, Camino et al. [13] presented a feature engineering detection method. They extracted 434 features by analyzing the differences between benign smart contracts and honeypots in code, transaction, and cash flow features. Besides, a machine learning model based on the XGBoost algorithm [14] was constructed to classify contracts. The classification results revealed 57 new honeypot instances, including two types of honeypots different from existing honeypots.

However, exploring how to defend existing honeypots to fight against honeypots is insufficient. The new honeypots that may appear in the future must also be predicted from the perspective of the attacker. Thus, an adversarial honeypot is proposed in this work. We obtain the code and behavior features of honeypots through the comparative analysis of 158,568 non-honeypot contracts and 352 honeypots. Besides, these features are hidden to form an adversarial honeypot, enabling the honeypot to bypass the detection of existing detection technologies. During the experiment, we summarize the essential and non-essential features of the honeypot. The cost of bypassing the detection is also measured. The proposed honeypot can effectively counter the existing detection technology and bypass the detection. The main contributions of this work are described as follows:

1) A kind of adversarial honeypot is proposed on the basis of condensing the essential features of honeypots.
2) The detection system is countered by disguising account, code, and transaction features, and the cost of countermeasures is quantitatively evaluated.
3) We construct 18 adversarial honeypots, and the ability of adversarial honeypots to evade detection is verified on a dataset of the order of hundreds of thousands.

2 Principles and Features of Honeypots

Smart contract honeypot is an effective attack method based on the blockchain smart contract technology. The popularity of honeypots poses a severe threat to the security of property on the blockchain. The working mechanism of honeypots must be understood, the development trend of honeypots must be illustrated, and approaches for defending against honeypot attacks must be established. First, we need to understand the lifecycle of the smart contract in-depth. On this basis, the attack intention of the honeypot is taken as the starting point to analyze the abnormality of the honeypot in the statistical features of various dimensions.

2.1 Overview of Smart Contract Lifecycle

The lifecycle of a smart contract on Ethereum would go through three stages: contract creation, contract call, and contract destruction.

Contract creation: EOA A creates a smart contract by constructing a transaction to the destination 0×0. The executable bytecode of the contract is stored in the data field of the transaction. Then, EOA A broadcasts the transaction to the network. The miner receiving the transaction verifies the validity of the transaction, the correctness of the format, and the legality of the transaction signature. With the successful mining of node C, the node packs the above-mentioned transactions together with other transactions into a block and broadcasts the block in the network. After the block broadcast is received by node C, other nodes would verify the block. If the verification is successful, then the block will be added to the local blockchain. The contract is successfully created, and the contract account address is also calculated from the address of EOA A and the nonce of transactions.

Contract call: Contract call includes two types: normal transaction call and internal transaction call. In Ethereum, smart contracts written by Solidity [15] generate a string of hexadecimal bytecodes after being compiled. During the process of calling a contract, the called function name and parameters must be converted into a string of hexadecimal byte codes and written into the transaction. The hexadecimal code will be filled in the data field of the transaction when the user creates or calls a smart contract. The subsequent process is the same as an ordinary transaction, and the entire network is synchronized after network propagation, mining, block broadcasting, and block verification.

Contract destruction: Contract destruction is an operation initiated only by the creator of the contract. The balance of the contract can be sent to the specified address by calling *selfdestruct (address owner)*, and the contract is destroyed. Once the contract is destroyed, interaction cannot continue, and the Ether transferred into the contract cannot be transferred out.

2.2 Principle of Honeypots

According to the differences of the internal working principles, honeypots can be divided into ten types: balance disorder, inheritance disorder, skip empty string literal, type deduction overflow, uninitialized structure, hidden status updates, hidden transfer, straw man contract, unexecuted calls, and map key encoding trick.

Balance Disorder (BD): The balance disorder honeypot has the following business logic:

Code 1. BD

```
1. contract A{
2.    function trans( ) public payable{
3.       if(msg.value >= this.balance){
4.          msg.sender.transfer(this.balance + msg.value;}
5.    }
6. }
```

Every smart contract in Ethereum has a parameter named balance. The parameter balance could be called in the form of *this.balance*. When the smart contract calls a function and performs a transfer operation, the modification operation of *this.balance* is always executed before the code in the function. However, some new users of Ethereum do not know it. The attacker takes advantage of this fact to make these users mistakenly believe that the transferred ether and the entire balance of the contract account will be transferred to their accounts as long as sending ether larger than the existing balance of the contract account. When the user calls the function *trans* and transfers to the contract, the value of *this.balance* has been updated, and *msg.value* can never be higher than *this.balance*. Therefore, the fourth line in Code 1 will never be executed.

Inheritance Disorder (ID): When a smart contract inherits from another smart contract, only one contract is created on the blockchain, and all the codes of the parent smart contract are copied to the created child smart contract. However, not every attribute in the parent smart contract will be inherited. When the child smart contract has a property with the same name as the parent smart contract, the property is actually new and has no relationship with the same name property of the parent class. In other words, the variables in the child contract cannot cover the variables of the same name in the parent contract. This problem still exists in the latest version. The attacker can construct the restriction conditions for withdrawals based on the attributes of the same name by using this feature. Accordingly, the victim mistakenly believes that the withdrawal conditions can be met by calling a certain function and sending some funds. Then, all funds in the contract account would be withdrawn. As illustrated in Code 2, contract B inherits from contract A. We notice two problems:

1) Function *b2* only allows the account that meets the owner to withdraw the balance;
2) Function *b1* can modify the value of the owner account.

The victim may try to modify the owner by calling function *b1* and then call function *b2* to withdraw the balance while failing to achieve this operation. This situation occurs because the owner parameter modified by calling function *b1* actually belongs to contract B. The *owner* parameter of contract A will not be overwritten by this operation, and the condition of the modifier *onlyowner* cannot be satisfied. Consequently, the withdraw operation will not succeed.

Code 2. ID

```
contract A{
  address public owner;
  owner = msg.sender;
  modifier onlyowner{if(msg.sender! = owner) revert(); _;}
}
contract B is A{
  address public owner = msg.sender;
  function b1( ) public payable{
   if(msg.value = this.balance)
   {owner = msg.sender;}
  }
  function b2( ) public onlyowner{
   msg.sender.transfer (this.balance);
  }
}
```

Skip Empty String Literal (SESL): An error previously included in the Solidity compiler can skip hard-coded empty string literals in the parameters of the function.

Code 3. SESL

```
contract A{
  function test(uint amount, bytes32 msg, address sender, address owner){
   transfer(amount, msg, sender, owner);
  }
  function trytest(uint amount){
   this.test(amount, ‘’, msg.sender. owner_);
  }
}
```

As indicated in Code 3, the second parameter *msg* is hard-coded as an empty string when the function *Trytest* calls the function *test*, causing the compiler to skip the empty string literal during compilation and move the following parameters forward. Accordingly, *msg.sender* is assigned to the *msg* parameter, *owner_* is assigned to the *msg.sender* parameter, and the default value of the *owner* parameter is zero. Therefore, the *test* function performs the balance transfer to the owner instead of *msg.sender*.

Type Deduction Overflow: The Solidity compiler supports type deduction. The compiler uses type deduction to automatically infer the smallest possible type when a variable is declared as var type. However, type deduction may cause an integer overflow, resulting in an endless loop and making the subsequent code impossible to execute [16].

Code 4. TDO

```
contract A{
  function Test() public payable{
    if(msg.value > 0.1 ether){
      uint256 multi = 0;
      uint256 amount = 0;
    for(var i = 0; i < msg.value * 2; i++){
      multi = i * 2;
      if(multi < amount){break;}
      amount = multi;
    }
    msg.sender.transfer(amount);
    }
  }
}
```

As shown in Code 4, the variable *i* in the for loop is declared as var type, and the editor infers that the type is uint 8. When *msg.value * 2* is greater than the maximum value of uint 8, the integer overflow causes an endless loop and the transfer operation cannot be successful.

Uninitialized Structure (US): The structure type can be customized in the smart contract. When calling these structures, data may be overwritten if they are not properly initialized.

Code 5. US

```
contract A{
  uint random = keccak256(abi.encode(now));
    struct Test{
        address public player;
        uint256 number;
    }
    function guess(uint256 guessnumber) public{
        test test1;
        test1.player = msg.sender;
        test1.number = guessnumber;
        if(guessnumber == random){
          msg.sender.transfer(this.balance);
    }
}
```

Code 5 exhibits a guessing contract. To withdraw funds from the contract account, the user needs to guess the value of *random*. Given that all data on the blockchain is transparent, the *random* can be easily determined. However, the structure *Test* is not initialized with keyword *new*. Accordingly, the address of the first variable of the contract is mapped to the address of the first variable of the contract which is variable *random*. Assigning a value to the *player* variable of *test1* will overwrite the original value of *random*, causing the judgment condition *guessnumber* == *random* to result in False and withdrawal failure.

Hidden State Update (HSU): Transaction with empty transaction value will not be displayed on Etherscan for the sake of brevity [17]. Taking advantage of this feature, the attacker would construct code logic and use the value of certain variables (taking *varSet* as an example for variable values) as the basis for judging whether the user can withdraw funds from the contract, as shown in Code 6. Inexperienced victims cannot find evidence that *varSet* is set to True and would assume the default value of False. However, attackers will abuse the above-mentioned features of Etherscan and modify the value of *varSet*, making the victim unable to withdraw funds.

Code 6. HSU

```
contract A{
  bool varSet = false;
  function SetPass(bytes32 hash) payable{
    if(!vaeSet && (msg.value >= 1ether)){hassPass = hash;}
  }
  function GetGift(bytes pass) returns(bytes32){
          if(hassPass == sha3(pass)) {msg.sender.transfer(this.balance);}
          return sha3(pass);
  }
  function PassHasBeenSet(bytes32 hash){
    if(hash == hashPass) {varSet = true};
  }
}
```

Hidden Transfer (HT): After attackers release the honeypot, they often promote it on certain platforms, such as Github, to attract the attention of the victim. Nevertheless, the Github display code can only have a certain width. Long codes are hidden and can only be seen by scrolling horizontally, as shown in Code 7. Attackers deceive the victim and prevent the balance from being transferred to the victim's account by constructing a long space to hide the subsequent code. This type of honeypot is called hidden transfer.

Code 7. Hidden Transfer

```
contract A{
  function Test( ) payable{
    require(msg.value > 0.5 ether);
  if(block.number > 1){if(_owner == msg.sender){_owner.transfer(this.balance);}
    msg.sender.transfer(this.balance);
  }
}
```

Straw Man Contract (SMC): The straw man contract involves two contracts: A and B. Two steps are needed to complete the recording function: initializing a contract B in contract A and calling the function of contract B. Then, EOAs are allowed to obtain the right to withdraw large amounts of funds from contract A after investing a small amount of funds in that contract. In actual operation, contract A does not initialize contract B, that is, seen by the victim while initializing a contract with the same name. Calling the contract with the same name would cause a rollback. Accordingly, the victim will be unable to successfully withdraw funds from contract A after investing a small amount of funds into it.

Code 8. SMC

```
contract A{
  function pay() public payable{
    if (msg.value > 1ether){owner = msg.sender;}
  }
  function setB(_b){B b = B(_b);}
  function withdraw(uint_val) public payable{
    require(msg.sender == owner);
```

```
        b.addmsg(msg.sender, _val, "add");
        msg.sender.transfer(_val);
    }
}
contract B1{
    struct Message{...}
    message[ ] public History;
    message Lastmsg;
    function addmsg(address_addr, uint_val, string _data){...}}
contract B2{
    struct Message{...}
    message[ ] public History;
    message Lastmsg;
    function addmsg(address_addr, uint_val, string _data){revert( );}
}
```

As illustrated in Code 8, the attacker constructs three contracts: A, B1, and B2. B1 and B2 have the same name and provide similar functions. When the attacker creates contract A, he calls the *setB* function to initialize a contract named B with an address. The victim mistakenly understands that the attacker calls the B1 contract; however, he actually calls the B2 contract. Calling the *addmsg* function of this contract will lead to a rollback, resulting in the failure of the withdrawal operation.

Unexecuted Call (UC): As presented in the following code, when the victim transfers more than 0.1 ether to the contract, the function call will be called to transfer the funds to the victim's account. However, in the actual situation, the function called lacks the necessary parentheses for execution when the victim calls the function *trans* to transfer funds from the contract account. Consequently, the call is not properly executed. In fact, *msg.sender.call.value(1 ether)( )* is correct way to call.

Code 9. UC

```
contract A{
  ...
  function trans() public payable{
    if(msg.sender.value >= 0.1 ether){
      msg.sender.call.value(1 ether);}
  }
}
```

Map Key Encoding Trick (MKET): Honeypots based on MKET use a mapping method to store *owner* variables. As indicated in Code 10, the contract uses "*Stephen*" as the key to access the *owner*. This key is used when initializing the contract and reading the *owner* variable. Besides, *msg.sender* is stored using "*Stephen*" as the key-value when calling the *becomeOwner* function. However, the third letter *e* that stores the key-value of *msg.sender* is in Cyrillic. Accordingly, the key-value is inconsistent with the key-value of the stored *owner* variable. The victim believes that the *owner* has been overwritten, and only the real contract creator can retrieve the funds stored in the contract account.

Code 10. MKET

```
contract BankOfStephen{
   mapping(bytes32 => address) private owner;
   constructor() public{
      owner['Stephen'] = msg.sender;
   }
   function becomeOwner() public payable{
      require(msg.value >= 0.25 ether);
      owner['Stephen'] = msg.sender;
      }
   function withdraw() public{
      require(owner['Stephen'] == msg.sender);
      msg.sender.transfer(this.balance);
   }
   function() public payable {}
}
```

2.3 Honeypot Feature Analysis

Honeypot is a special type of smart contract. This contract uses seemingly obvious flaws to attract the victim to attack, causing the victim's digital currency to flow to the honeypot. This digital currency cannot be retrieved and can only be withdrawn by the contract creator or designated account [18]. A smart contract is composed of its contract account, contract code, and transaction. The attacker needs to execute the following steps to complete the attack:

Step 1: Construct a smart contract with seemingly obvious flaws.

Step 2: Use EOAs to create smart contracts and transfer initial funds to the smart contract as bait.

Step 3: Attract the victim to transfer money to the contract account and withdraw the funds in the contract account as income. The abnormality of honeypots is reflected in the three dimensions of account, code, and transaction.

Account features: The accounts associated with the smart contract include a creator account and a transaction account. From a profit point of view, attackers are inclined to deploy honeypots in batches to increase revenue. Therefore, multiple honeypots may be deployed by the same account, and the deployment time of honeypots is generally close. If the attacker holds multiple accounts, then the creator account of the honeypot and the transaction account may overlap.

Fig. 1 illustrates the relationship among the contract, creator, and transaction accounts of eight types of honeypot contracts. UC and MKET are not counted because few instances of both have been discovered to support the analysis. In the figure, the pink node represents the contract account, the green one indicates the creator account, and the blue one denotes the transaction account. Moreover, the dataset of each subgraph is all the currently known contracts of the corresponding type of honeypot.

The figure demonstrates that the longest paths of the graphs of BD, US, HSU, HT, ID, SESL, and SMC are distributed in the interval of [3,16], and EOAs with node degrees greater than one are found in all eight honeypots. The longest path equal to two indicates that the creator and transaction accounts of each honeypot contract are independent and do not overlap with the creator or trading accounts of other honeypot contracts. The length greater than two suggests that the creator or transaction account of one honeypot contract serves as the creator or transaction account of another honeypot contract. The degree of an EOA node greater than one demonstrates that the EOA has participated in the creation or transaction process of more than one honeypots.

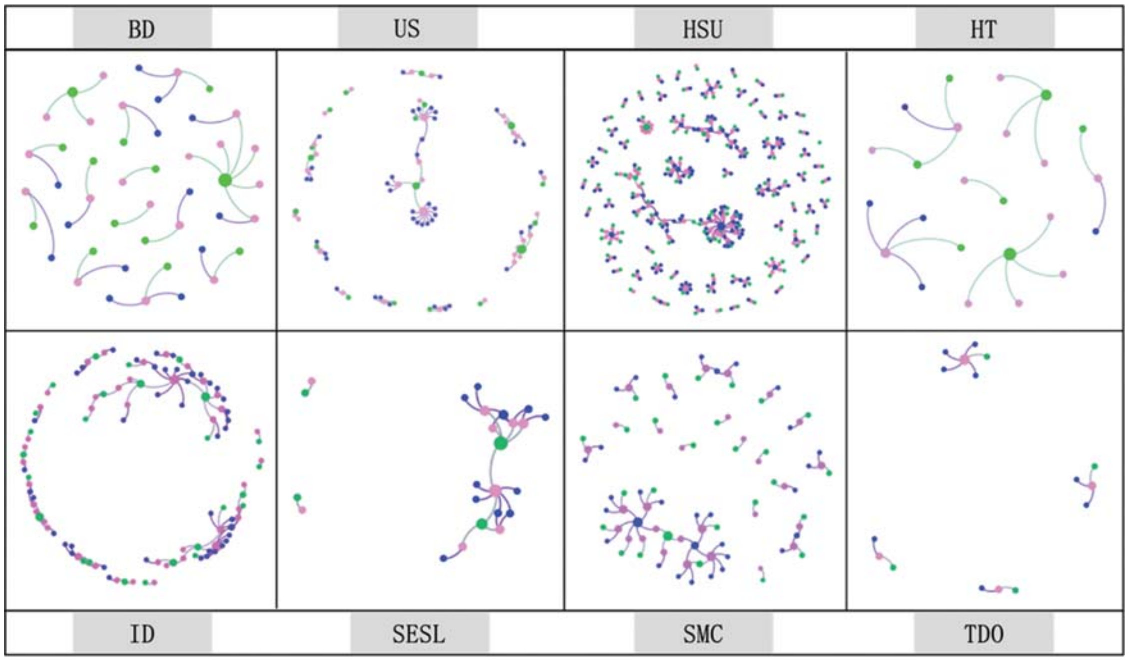

Figure 1: Account relationship of honeypots

In summary, the two situations are as follows: First, the same account would create multiple honeypots of the same type. Second, the same account may participate in the creation of one honeypot and the transaction of another honeypot.

Code features: The honeypot code constructed by the attacker is relatively short to guide the victim to transfer funds to the contract account, facilitating the victim to effectively find the vulnerabilities in the code. In this study, we count the codes of 352 known contract honeypots. The results demonstrate that the code length of the honeypot contract is within two hundred lines with an average of 57 lines. The average number of code lines in non-honeypot contracts is 279 lines. More than 30% of non-honeypot contracts have more than two hundred lines of code. Besides, most compiler versions of honeypots discovered to date are 0.4.19. This situation may be related to the peak popularity of Ethereum.

Transaction features: The transaction features of honeypots are abnormally reflected in the transaction amount, gas fee, cash flow, and contract lifecycle. The average and variance of the transaction amount of honeypots are smaller compared with non-honeypots regardless of the decoy deployment, induced sending of funds, or attacker withdrawing funds. In terms of the gas fee, certain honeypot contracts use the re-entry vulnerability to attract victims. This vulnerability exploits the *call.value( )* function to generate extremely high gas costs. With regard to the cash flow, 90.17% of known honeypots contain cash flows from the creator's account to the contract account; 36.61% of known honeypots contain cash flows from the contract account to the creator account; the two accounted for 52.70% and 3.76% in non-honeypot contracts, respectively.

The interval from the creation of a smart contract to the last transaction is called the contract lifecycle. The contract lifecycle of honeypots is generally short.

In summary, an effective honeypot abnormally performs in the three dimensions of account, code, and transaction, as presented in the following table.

Table 1: Honeypot features

Feature dimension	Feature No.	Description
Account features	1	The same account deploys a large number of smart contracts in a period of time
	2	The account deploying the contract has deployed a honeypot contract
	3	The account deploying the contract has a transaction with the honeypot contract account
Source code features	4	Number of code lines
	5	Compiler version used by the source code
Transaction features	6	Fund flows
	7	The average transaction amount
	8	The variance of the transaction amount
	9	Transaction gas consumption
	10	Number of transactions on contract account
	11	The lifecycle of smart contract

3 Principle of Adversarial Honeypot

The key problem in constructing a honeypot with antidetection capabilities is to hide the abnormal features of the honeypot. This chapter first analyzes the concealability and hiding methods of honeypot features from the three dimensions of account, source code, and transaction. Then, the construction method of adversarial honeypots is summarized. Finally, this chapter discusses automatic countermeasures against honeypot detection tools.

3.1 Analysis of the Confrontation of Features

3.1.1 Account Features

Honeypots are short and easy to replicate. Attackers often deploy honeypot in batches to obtain high profits. We analyzed existing honeypots and found that 13.82% of honeypots are present, whose creator accounts have deployed more than one honeypot. Approximately 17.24% of honeypots are present, whose transaction accounts have transactions with other honeypots at the same time. The correlation between the account and the honeypot must be reduced to improve the confrontation of honeypots. We use two methods to reduce the abnormality of honeypot features in account dimensional: 1. Use the anonymity of the blockchain to register multiple accounts and deploy honeypots with different accounts; 2. Honeypot creator and transaction accounts do not participate in other honeypot transactions.

3.1.2 Code Features

The number of code lines of known honeypots is distributed in the interval [19, 185], and the average number of code lines is 22.8. Meanwhile, the number of code lines of non-honeypot contracts is distributed in the interval [1, 11409], whose average number is 278.4. The above data can show that the feature distribution of honeypots and non-honeypots is quite different in terms of the number of lines of code. If the number of lines of the honeypot is increased, then the indistinguishability between the honeypots and the non-honeypots can be greatly increased. Functions that do not affect the attack function, such as log functions, can be added while maintaining the honeypot attack function to achieve the above purpose. However, the number of lines of code in a honeypot contract should not be too much. First, the victim will have difficulty finding the vulnerability deliberately set by the attacker owing to the many lines of code. Second, too many lines of code will increase the cost of contract deployment.

The compiler version has less impact on determining whether a contract is a honeypot compared with the number of code lines. However, statistics show that most honeypot contracts are compiled with version 0.4.19, which is most likely a concentration of honeypot contracts in a short time. Using the latest version of the compiler to construct a honeypot is advantageous to avoid detection.

3.1.3 Transaction Features

Transaction features are the prominent features of honeypots. These features include transaction amount, transaction frequency, fund flow, and the life cycle of the contract. The following analyzes the concealability of these features one by one and proposes hiding methods for the concealable features.

Transaction amount: The transaction amount of honeypots is distributed in a small range. Statistics show that the transaction amount range of the known honeypots is [0, 1.67] ether, and the average transaction amount is 0.27 ether; 50% of the transactions are distributed between [0.1, 0.38] ether, and 25% of the transaction amount is [0.38, 1.67] ether. The range of transaction amount of non-honeypot contracts is [0, 204365.82] ether, with an average value of 5.46 ether. Nevertheless, more than 75% of the transaction transfer funds are zero. These data indicate that honeypots have more transactions with non-zero funds transferred compared with non-honeypot contracts. However, the transaction amount of non-zero transactions is smaller. The transaction amount distribution of the honeypots should be constructed to blur the difference between honeypots and non-honeypots. Accordingly, the average value or variance of honeypots could be close to the distribution of non-honeypots.

The number of transactions: A honeypot is a smart contract constructed by an attacker to attract victims to invest funds. The number of victims who are successfully attracted by the honeypot is limited, and the same one will not be cheated multiple times. Accordingly, the number of honeypot transactions is generally small. The number of transactions of known smart contract instances ranges from 1 to 32, and that of non-honeypot contracts ranges from 1 to tens of millions. The average number of transactions and variance of honeypot contracts are smaller compared with non-honeypot contracts; thus, smart contracts with fewer transactions are easier to be classified as honeypots than smart contracts with more transactions. The number of honeypot transactions must be increased as much as possible to disguise honeypot transactions.

Fund flow: In honeypot detection based on feature engineering, the fund flow features are powerful features for identifying honeypots. In the work of Camino et al. [13], the influence of various features on distinguishing honeypots from non-honeypots was measured. Deposits from contract creators are an influential feature, with an influence of 0.657, while the influence of the second is only 0.107. Illustrating the importance of this feature is sufficient. In addition to this feature, the investment from the non-creator and the withdrawal of the contract creator are also features that have greater effects. Given that the basic purpose of honeypots is to extract funds from the victim, investment from non-creators is inevitable. This feature is the most essential feature of successful honeypots. In terms of the deposit from the contract creator and the withdrawal from the contract creator, we propose an investment honeypot, which introduces the role of investors in the honeypot for decoy deployment and fund extraction and hides the abnormal flow of funds with investment thinking.

Life cycle: Among all known honeypots, only approximately 35% of honeypots successfully launched an attack. Specifically, 65% of honeypots no longer have new ones from non-creators after the decoy is constructed. In this case, many creators choose to withdraw funds or even destroy the honeypot. Accordingly, the life cycle of honeypots is generally relatively short. To disguise a honeypot, a transaction can be made with the honeypot contract after a random period to extend its life cycle.

Summarizing the concealability and hiding methods of the features of honeypots are shown in Tab. 2. The feature numbers correspond to the features in Tab. 1. Tab. 2 illustrates that most abnormal features of known honeypots are not the essential features of honeypots and can be hidden by corresponding means, in addition to the features of fund flow.

Table 2: Concealability and concealment methods of honeypot features

Dimension of features	Feature No.	Essential feature or not	Concealable or not	Concealment method
Account	1	No	Yes	Use different accounts to deploy honeypots
	2	No	Yes	Use different accounts to deploy honeypots
	3	No	Yes	The honeypot creator account does not participate in transactions of other honeypots
Code	4	No	Yes	Increase the number of lines of code
	5	No	Yes	Use the latest compiler version
Transaction	6	No	Partially concealable	Investment honeypot
	7	No	Yes	Construct the amount of transactions
	8	No	Yes	Construct the amount of transactions
	9	No	Yes	Construct transactions to increase the number of honeypot transactions
	10	No	Yes	Extend the life cycle of the honeypot

3.2 Construct an Adversarial Honeypot

Goodfellow et al. [19] put forward the concept of Adversarial Examples in a paper published by ICLR. It represents input samples formed by deliberately adding subtle interference in the data set. The input after the interference causes the model which has a high degree of confidence to give an incorrect output. We refer to adversarial examples based on smart contract honeypots as adversarial honeypots.

The purpose of constructing an adversarial honeypot is to study the feasibility of using a known honeypot technology to bypass honeypot detection tools through camouflaging. The structure of the adversarial honeypot should be based on the known honeypot technology and strive to achieve the purpose of bypassing detection through a few changes. The ten known honeypot technologies essentially attract the victim by constructing seemingly obvious vulnerabilities. The difference lies in the seemingly obvious vulnerabilities of the structure. The detection method based on feature engineering extracts universal features and does not care about the vulnerabilities. Accordingly, these seemingly obvious vulnerabilities can be used when constructing adversarial honeypots. The transformation has a great effect on honeypot classification. On the basis of this idea, the construction steps of an adversarial honeypot are as follows:

The first step is account construction: create no less than two EOAs that have neither deployed honeypots nor participated in honeypot transactions as creators and transaction accounts.

The second step is code construction: expand the code function and number of lines based on the code of the known honeypot instances. The expansion of the code function depends on the investment honeypot. The specific implementation method is as follows:

1. Honeypot construction: Add an investment function under the premise that the code implementing the honeypot function is not affected. The function is: for the first EOA that calls this function to transfer funds to the contract account, mark its identity as an investor, and grant it the right to withdraw funds from the contract account. After the investment function is successfully called for the first time, calling it again will cause a rollback.

2. Honeypot contract deployment: Choose an EOA held by the attacker that is not involved in other honeypot transactions. Use this account to deploy the constructed code. The honeypot can be promoted offline to attract the victim to launch an attack.

3. Decoy fund transaction construction: Choose an EOA that is different from the contract creator from the EOAs held by the attacker, and call the investment function to invest in the contract. The investment amount is related to the purpose to attract the victim, and no upper limit exists.

4. Withdrawal: After successfully attacking, the investor or the creator calls the withdrawal function to withdraw proceeds.

The code example of the investment honeypot is shown in Code 11:

Code 11. Example of Adversarial Honeypot Code

```
pragma solidity ^ 0.8.0;

contract AdvContract {
  #construction of maybe-obvious-flaws
  #construction of real trap
  ...
  address public sponsor;
  bool initialized;
  function Sponse() public payable{
    if (initialized == false && msg.value > 0.1 ether){
      sponsor = msg.sender;
      initialized = true;
    }
    else{revert( );}
  }
  function withdraw() public {
    require(msg.sender == owner || msg.sender == sponsor);
```

```
        msg.sender.transfer(address(this).balance);
    }
    ...
}
```

The third step is transaction construction: the account that has not participated in the honeypot transaction provides the honeypot with funds used as bait and confirms the identity of the investor. Before the attack is successful, additional irrelevant transactions can be constructed to increase the number of transactions of the honeypot and construct the transaction amount based on the non-honeypot contracts, in addition to the transactions generated by the decoy. After the attack is successful, the investors or creators withdraw funds.

3.3 Automated Confrontation

The construction of adversarial honeypots can be automated. To improve efficiency and increase scalability, we propose automated countermeasures to empower the process of honeypot construction with deep learning. According to the degree of automation, the mechanisms can be divided into:

Automated feature generation: Feature generation with the help of artificial intelligence algorithms is a semi-automatic method for constructing honeypots. Models are constructed by deep learning algorithms, such as convolutional neural networks [20] or generative adversarial networks [21]. Feature vectors constructed from the dimensional features of non-honeypot contracts are used as the training set to train models. With parameter adjustment, the final output result is an equal-length vector with the same distribution as the feature vector of the training set. The honeypots that are not adversarial can be adjusted to make their features indistinguishable from non-honeypot contracts by using this vector as a benchmark. Therefore, the confrontation between adversarial honeypots and detection tools based on feature engineering can be realized.

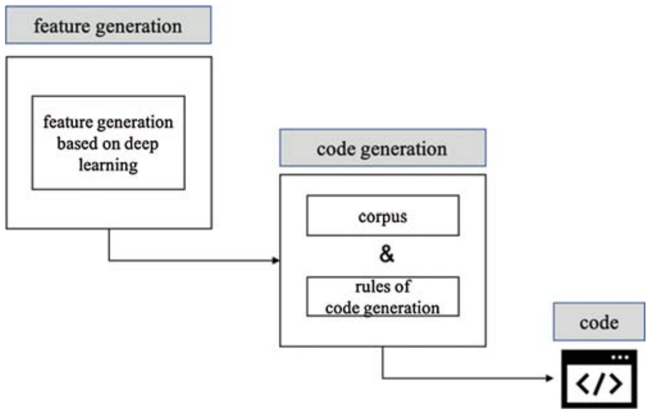

Figure 2: Automated code generation

Automated code generation: Automated code generation is based on automated feature generation to further realize the automation of code construction. The automation of code generation is realized by the feature generation module and the code generation module together (Fig. 2). The feature generation module uses a deep learning algorithm to generate feature vectors. The generated feature vectors fit the non-honeypot feature distribution and will be the input of the code generation module. The code generation module needs to construct a *corpus* from which the honeypot type, names of variables, function order, account creation, trading account, and other features are selected. The code comes from the *corpus* and is combined according to the code generation rules. Then, a creation account is selected to deploy the contract, and a series of transactions is created according to the generated feature vectors to achieve the purpose of setting a bait.

4 Experiment

4.1 Dataset and Experimental Setup

This study analyzes 158,568 non-honeypot contracts and 352 honeypot contracts and summarizes the feature abnormalities of honeypots. A kind of adversarial honeypot is proposed on the basis of condensing the essential characteristics of honeypots. We have constructed 18 adversarial honeypots based on the ten known honeypot technologies. The dataset used in this experiment is shown in Tab. 3.

Table 3: Dataset

Dataset	Contract type	Numbers
Benign dataset	Non-honeypot	158,568
Honeypot dataset	Honeypot	352
Adversarial honeypot dataset	Adversarial honeypot	18

The honeypot dataset contains honeypots based on the ten types of technologies, and the adversarial honeypot dataset constructs adversarial honeypots on the basis of the honeypot dataset.

We use the detection tool proposed by Camino et al. to test the effectiveness of the adversarial honeypot. The detection tool is based on feature engineering. It extracts the code, transaction, and fund flow features of the honeypot, and trains the XGBoost classifier against the honeypot. We download the open-source program [22] on Github and run it to directly to detect the dataset.

The experiment is divided into three groups. The first group is the control group. The model is trained with the benign and honeypot datasets, and the classification effect of the model on existing honeypot contracts is obtained. The second group is the experimental group. First, we evaluate the classification effect of the model trained in the control group on the adversarial honeypot. Then, we use the adversarial honeypot dataset to retrain the model and evaluate the classification effect on the adversarial honeypot again. The third group is the quantitative evaluation group, which quantitatively evaluates the countermeasure cost required to bypass the detection.

The experimental results are measured by accuracy and recall. The calculation methods of the two are as follows:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

And the meaning of TP, FP, TN, FN are shown in Tab. 4.

Table 4: Confusion matrix

Actual	Predict		
		Yes	No
	Yes	TP	FN
	No	FP	TN

4.2 Experimental Evaluation

(1) Control group: The experiment of the control group trained a machine learning model on the basis of XGBoost to classify the contract as honeypot or non-honeypot. The experiment uses the k-fold cross-validation [23] for fitting and enhancing the generalization ability. The dataset is divided into eleven parts according to the honeypot type, and one of them is taken as the test set, and the other ten parts are used as the training set. In the experiment, honeypots are marked as positive samples, and non-honeypots are labeled as negative samples. The experimental results are shown in the following figure.

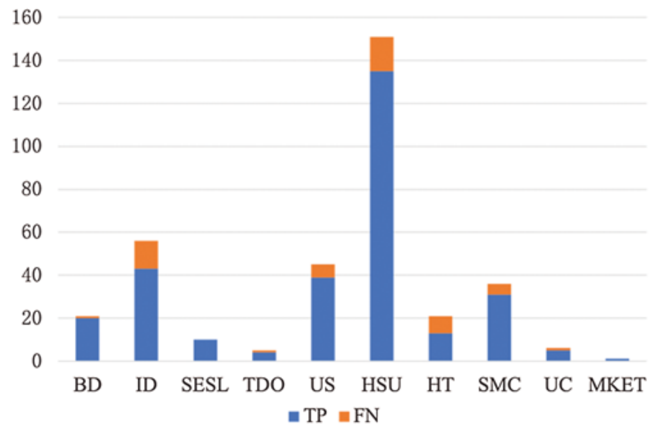

Figure 3: Classification result of the control group

Fig. 3 shows the classification results of the model for ten honeypot technologies currently known. The blue part represents TP and the orange part represents FN. The proportion of FN in the figure is much smaller than that of TP, indicating that the probability of honeypot contracts being discovered by the detection model is extremely high. Specific to different types of honeypots, the accuracy of the model for BD, SESL, and MKET is higher than the other seven types. Overall, the accuracy of the model is 98.78%, and the recall rate is 95.25%.

(2) Experimental group: The experimental group is set up to evaluate the adversarial ability of adversarial honeypots. We set two groups of experiments. The first group uses the model trained in the control group to predict 18 adversarial honeypots; the second group adds five adversarial honeypots to the training set. Oversampling is used in this study due to the small dataset of adversarial honeypots. We train a new machine learning model by using the training set added with adversarial honeypots, called a retraining model.

Tab. 5 shows the results of the control and experimental groups. The control group tested the normal honeypots. The result shows that the undetected number was 17, and the recall rate was

95.25%. The first experimental group uses the control group model to detect adversarial honeypots. The result shows that the undetected number is 18, and the recall rate is zero, indicating that the generalization ability of existing models is insufficient, and adversarial honeypots can bypass the detection. The second experimental group retrained the model and then detected adversarial honeypots. The undetected number was 12, and the recall rate was 7.69%, thus indicating that the retraining model has improved its ability to detect adversarial honeypots, but it still has a large part of adversarial honeypots that can escape detection.

Adversarial honeypots can effectively counter the detection based on feature engineering.

Table 5: Experimental result

Experimental group	Honeypot types	Number of honeypots	TP	FN
Control group	Normal honeypot	352	335	17
First experimental group	Adversarial honeypot	18	0	18
Second experimental group	Adversarial honeypot	13	1	12

The experimental group also measured the account relationship distribution of adversarial honeypots Fig. 4. Part A shows the relationship distribution of normal honeypot accounts, and that in Part B exhibits the relationship distribution of the adversarial honeypot accounts. Several roads can be observed in the ordinary honeypot account relationship graph with a length greater than two, which demonstrates the presence of a large number of honeypots with overlapping creator transaction accounts. The length of the longest path in the adversarial honeypot account relationship graph is two. The creator accounts and transaction accounts representing all honeypots will not overlap. The independence between the adversarial honeypot accounts can counter the honeypot detection technology on the basis of the graph mining algorithms compared with normal honeypots.

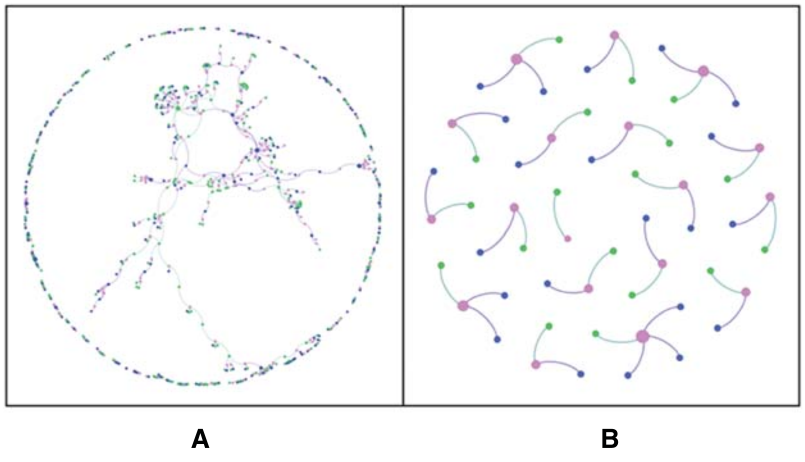

Figure 4: Account relationship before and after confrontation

(3) Quantitative evaluation group: This group of experiments aims to explore the cost of counter-detection tools. First, the confrontation cost is divided into two categories: fund costs

and construction costs. With regard to the cost of funds, we mainly explore three questions. The first question is the smallest bait that successfully attracts the victim. The second question is the smallest transfer of funds based on the smallest bait that successfully opposes the detection tool. The last one is the minimum number of transaction that successfully opposes the detection tool. In terms of the construction cost, we will explore the cost of the number of lines of code, the cost of the number of accounts, and the cost of construction time. The quantitative evaluation experiment follows the principle of control variates, taking the parameter to be explored as the variable, keeping the parameter that has no causal relationship with the constant variable and obtaining the final experimental result.

Fund cost: We count the transactions of honeypot instances, and the data shows that the minimum bait that can successfully attract the victim is 0.001875 ether. The minimum fund transfer based on this amount is composed of bait and transaction fees. The transaction fees are related to the volume of deployed contracts and the complexity of calling functions. With regard to honeypots with generally low complexity, the transaction fees are distributed in the range of [21000 wei, 8000000 wei].

To evaluate the minimum number of transactions for successfully countering detection tools, we kept the code of the adversarial honeypot and the creator account, and tried to deploy the decoy with the creator account, one transaction account, and multiple transaction accounts. Accordingly, the minimum number of transactions to successfully counter the detection tool is two, one is to create a contract, and another is to deploy a decoy.

Construction cost: With the principle of control variates, we evaluate the number of lines of code, the number of accounts, and the construction time of the adversarial honeypot that can successfully counter the detection. The number of lines of code that can bypass the detection of different honeypots is inaccurate without modifying the main code that constitutes the honeypot. However, the shortest number of lines is distributed in the interval [40, 60]. Under the ideal conditions of code and transaction features, even if the creator deploys a bait, or the creator withdraws funds, it can successfully counter detection. The time it takes to construct a honeypot has little effect on the resistance of the honeypot.

5 Conclusion

This study analyzes the data of 158,568 non-honeypots, 352 normal honeypots, and 18 adversarial honeypots. We propose a type of adversarial honeypot based on the essential features of honeypots. This honeypot takes account, code, and transaction features to counter the detection system. The experimental result shows that the success rate of the adversarial honeypot is 100%. After the detection model with adversarial honeypots is retained, the success rate of the adversarial honeypot is 92.31%. This result indicates that the ability of adversarial honeypots to escape detection is still stronger than normal honeypots even under the premise of captured samples. We compared the account relationship between normal and adversarial honeypots. The result shows that the account relevance of different instances of adversarial honeypots is greatly reduced, which is beneficial to countering the honeypot detection technology on the basis of the graph mining algorithms.

This study also focuses on evaluating the cost of constructing adversarial honeypots. The fund costs, such as the minimum decoy amount, the minimum transfer funds, and the minimum number of transactions for constructing adversarial honeypots, and the construction costs, such as the minimum number of lines of code, the minimum number of accounts, and the shortest construction time, are measured through experiments with the principle of control variates. The experimental result shows that the cost of the attacker's construction of adversarial honeypots is extremely low compared with the possible benefits, and most fund costs can be recovered. We also propose an automated countermeasure method for honeypot detection tools, which uses artificial intelligence to empower the process of antagonistic honeypot construction, compared with the possible benefits. The automated countermeasure method can effectively improve the antagonistic ability of adversarial honeypots and greatly save the labor of construction. Therefore, the automated method is suitable for the scenario of mass deployment of honeypots. In the future, adversarial honeypots may appear on a large scale and endanger the safety of the property on the blockchain, and defensive countermeasures should be proposed as soon as possible.

Funding Statement: This work was supported by the Guangdong Province Key Field Research and Development Program under Nos. 2019B010137003 and 2019B010136004, the National Key Research and Development Program under Grant Nos. 2018YFB0803504 and 2019YFA0706404.

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

References

1. Buterin, V. (2014). A next-generation smart contract and decentralized application platform. *White Paper, 3(37),* 1–36.
2. Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper, 151,* 1–32.
3. Nakamoto, S. (2009). *Bitcoin: A peer-to-peer electronic cash system*. Manubot.
4. Hara, K., Sato, T., Imamura, M., Omote, K. (2020). Profiling of malicious users using simple honeypots on the ethereum blockchain network. *IEEE International Conference on Blockchain and Cryptocurrency*, pp. 1–3. Toronto, Canada. DOI 10.1109/ICBC48266.2020.9169469.
5. Chen, W., Guo, X., Chen, Z., Zheng, Z., Lu, Y. et al. (2020). Honeypot contract risk warning on ethereum smart contracts. *2020 IEEE International Conference on Joint Cloud Computing*, pp. 1–8. Oxford, UK. DOI 10.1109/JCC49151.2020.00009.
6. Sherbachev, A. (2018). Hacking the hackers: Honeypots on ethereum network. https://hackernoon.com/hacking-the-hackers-honeypots-on-ethereum-network-5baa35a13577.
7. Cheng, Z., Hou, X., Li, R., Zhou, Y., Luo, X. et al. (2019). Towards a first step to understand the cryptocurrency stealing attack on ethereum. *22nd International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 47–60. Beijing, China.
8. Vasek, M., Moore, T. (2015). There's no free lunch, even using bitcoin: Tracking the popularity and profits of virtual currency scams. *International Conference on Financial Cryptography and Data Security*, pp. 44–61, San Juan, USA, Berlin, Heidelberg: Springer. DOI 10.1007/978-3-662-47854-7_4.
9. Yin, J., Cui, X., Liu, C., Liu, Q., Cui, T. et al. (2020). CoinBot: A covert botnet in the cryptocurrency network. *Proceedings of the 22th International Conference on Information and Communications Security*, pp. 107–125, Copenhagen, Denmark, Cham: Springer. DOI 10.1007/978-3-030-61078-4_7.
10. Atzei, N., Bartoletti, M., Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (SOK). *International Conference on Principles of Security and Trust*, pp. 164–186, Uppsala, Sweden, Berlin, Heidelberg: Springer. DOI 10.1007/978-3-662-54455-6_8.
11. Luu, L., Chu, D. H., Olickel, H., Saxena, P., Hobor, A. (2016). Making smart contracts smarter. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269. Vienna, Austria. DOI 10.1145/2976749.2978309.
12. Torres, C. F., Steichen, M. (2019). The art of the scam: Demystifying honeypots in ethereum smart contracts. *28th USENIX Security Symposium*, pp. 1591–1607. CA, USA.

13. Camino, R., Torres, C. F., Baden, M., State, R. (2020). A data science approach for detecting honeypots in ethereum. *2020 IEEE International Conference on Blockchain and Cryptocurrency*, pp. 1–9. Toronto, Canada. DOI 10.1109/ICBC48266.2020.9169396.
14. Chen, T., Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM Sigkdd International Conference on Knowledge Discovery and Data Mining*, pp. 785–794. San Francisco, USA. DOI 10.1145/2939672.2939785.
15. Dannen, C. (2017), *Introducing ethereum and solidity*, vol. 1. Berkeley: Apress. DOI 10.1007/978-1-4842-2535-6.
16. Torres, C. F., Schütte, J., State, R. (2018). Osiris: Hunting for integer bugs in ethereum smart contracts. *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 664–676. San Juan, USA. DOI 10.1145/3274694.3274737.
17. Etherscan (2019). Etherscan. https://etherscan.io/.
18. Siegel, D. (2016). Understanding the dao attack. https://www.coindesk.com/understanding-dao-hack-journalists.
19. Goodfellow, I. J., Shlens, J., Szegedy, C. (2014). Explaining and harnessing adversarial examples. arXiv preprint, arXiv: 1412.6572.
20. Kalchbrenner, N., Grefenstette, E., Blunsom, P. (2014). A convolutional neural network for modelling sentences. arXiv preprint, arXiv: 1404.2188.
21. Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B. et al. (2018). Generative adversarial networks: An overview. *IEEE Signal Processing Magazine, 35(1),* 53–65. DOI 10.1109/MSP.2017.2765202.
22. Camino, R. (2018). Rcamino/honeypot-detection. https://github.com/rcamino/honey- pot-detection.
23. Rodriguez, J. D., Perez, A., Lozano, J. A. (2009). Sensitivity analysis of k-fold cross validation in prediction error estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(3),* 569–575. DOI 10.1109/TPAMI.2009.187.