Pied-Piper: Revealing the Backdoor Threats in Ethereum Smart Contracts

Fuchen Ma
Tsinghua University

Meng Ren
Tsinghua University

Lerong Ouyang
The University of Hong Kong

Yu Jiang
Tsinghua University

Zhe Liu
Nanjing University of Aeronautics and Astronautics

Han Liu
Tsinghua University

Qi Wang
University of Illinois at Urbana-Champaign

Jiaguang Sun
Tsinghua University

Abstract

With the development of decentralized networks, smart contracts are attracting more and more Dapp users. There are some functions in smart contracts that could only be invoked by a specific group of accounts. Among those functions, some even can influence other accounts or the whole system without prior notice or permission. These functions are referred to as smart contract backdoors. The backdoors in smart contracts have caused a lot of property losses and have become great harm to user's privacy in recent years. However, examining the existence of backdoors in smart contracts is a tough issue for most users as only limited smart contracts deployed on Ethereum provide the source code.

In this work, we propose Pied-Piper, a static analysis technique to detect potential backdoors in smart contracts on bytecode level. First, through an empirical study, we classify 5 common types of smart contract backdoors. Then Pied-Piper uses control flow graph based datalog analysis to reveal the backdoor threats. We first evaluated Pied-Piper on 100 smart contracts manually labelled with different types of backdoors. It reported 98 backdoors without false positives, and 2 of the injected backdoors are missed. Then, we applied Pied-Piper on 13484 real contracts deployed on Ethereum. Pied-Piper reported 205 threats, with 189 backdoors confirmed and 16 false positive errors identified by the smart contract developers. On average, each contract takes 8.03 seconds for analysis, and the false-positive rate of which is relatively low compared to other static analysis based techniques and could be easily eliminated by manually confirmation.

1 Introduction

Ethereum is a decentralized platform which supports smart contracts. Users could write smart contracts in a high-level language such as Solidity [11] and deploy the contracts on the platform. The source code of the contracts will be compiled to low-level bytecodes and be translated and then executed by Ethereum Virtual Machine. Since its creation, Ethereum has attracted more and more users. There are approximately 9.4 transactions [12] per second nowadays on Ethereum and most of the transactions are financial concerned. So, it's essential to protect the transaction process from attacks, that is, to make sure that the smart contracts are free of vulnerabilities.

However, in recent years, property loss accidents caused by vulnerabilities in smart contracts are emerging in endlessly. Among the contract vulnerabilities, the threat related to backdoors is often overlooked and thus creates great potential risk for users' property and privacy. *Backdoors are a category of high-privilege functions, which can only be invoked by certain group of accounts (owner of the contract usually), and will result in a great effect on other accounts.* **In June 2018, one firm in Australia lost $6.6 million due to a backdoor function in Soarcoin contract [35,40].** This case arose widespread concerns and the public thus began to focus on the threat caused by smart contract backdoors. However, most of the smart contracts have no available source code [16], so it is difficult for users to judge whether there is a backdoor in the contract without directly examining the code.

Moreover, though there are many works aiming at detecting software backdoors and having yielded noteworthy results, most of them do not match this context due to the distinguished definition of backdoors. For example, the traditional definition of a backdoor in software usually refers to an approach exploited by attackers to cheat in or bypass the permission authentication process. While in deep learning systems, a backdoor always means the pre-designed samples injected to poison the dataset by which the attacker can compromise the whole process. Therefore, a specific definition for backdoor function in smart contracts along with the techniques to identify a backdoor function is needed. However, in a decentralized system, it is difficult to define whether the influence caused by one function is acceptable or not. As few relevant benchmarks are founded in this field, so setting a concise and precise backdoor definition is a vital task for detecting them.

In this work, we propose Pied-Piper, a static analysis technique that can automatically reveal the backdoor threats in Ethereum smart contracts. First of all, we classify and identify

5 common types of backdoors by an empirical study. The first type is *Arbitrary Transfer*, which permits the owner of the contract transferring any amount of tokens from any address to another address. The second backdoor is *Generate Token*, which allows the owner to generate any amount of tokens out of the void. The third one is *Destroy Token*, which refers to destroying any amount of tokens from any address. The fourth type is *Disable Transferring*, which could stop all accounts from transferring tokens. The last one is named *Freeze Account*, which could forbid all operations of any account. The 5 types of backdoors are summarized and derived from quantities of real contracts deployed on Ethereum. After the classification and definition of backdoors, Pied-Piper uses domain-specific Datalog analysis to identify threats. It will first decompile the contract bytecode to an intermediate representation, build the control flow graph (CFG) of the contract, and construct some facts based on it. Then, it analyzes the CFG to check whether some backdoor related constraints hold. In this way, Pied-Piper could detect the backdoors in smart contracts based only on the bytecode.

For evaluation, we implemented Pied-Piper based on Vandal [4], an analysis framework for the extraction of program properties. We first tested Pied-Piper on 100 contracts manually labelled with different types of backdoors. It reported 98 threats without false positives, and 2 of the injected backdoors are missed. Then, we applied Pied-Piper on 13484 real-world smart contracts' bytecode and found that **189 of the contracts had confirmed backdoors**[1]. Specifically, 1 contract has been found with *Arbitray Transfer*, 34 with *Generate Token*, 29 with *Destroy Token*, 29 with *Disable Transferring* and 95 with *Freeze Account*. It took 30 hours in total to analyze these contracts. Each contract took an average of 3.6s for Datalog analysis and 4.4s for decompilation. The results show that Pied-Piper is effective and efficient in revealing backdoors in real-world smart contracts. Through the manual check on the results of Pied-Piper, developers also identified 16 false-positive situations, which is relatively low compared to other static analysis based techniques. We made a thorough analysis on the reason of these false alarms and the missing alarms, and gave some possible solutions to these problems. Overall, our work makes the following contributions:

- We classified and identified 5 common types of smart contract backdoors. To the best of our knowledge, we are the first to give a specific definition of smart contract backdoors by making an empirical study to summarize this kind of vulnerabilities.
- We designed and proposed Pied-Piper, the first static analysis tool that could automatically analyze whether a smart contract has a backdoor on the bytecode level.
- We implemented Pied-Piper and conducted several experiments to show the effectiveness of Pied-Piper. With Pied-Piper, we have found 189 confirmed backdoor threats in 13484 real-world smart contracts.

The rest of the paper is organized as follows: Section 2 presents a motivating example and how Pied-Pier works on this example. Section 3 introduces the background information about Ethereum smart contracts and Datalog analysis. Section 4 describes the 5 common types of smart contract backdoors in detail. Section 5 officially introduces the working pipeline of Pied-Piper. Section 6 presents the implementation details and gives the evaluation of Pied-Piper. Section 7 lists several related work. Section 8 concludes this work and describes some ideas about future work.

2 Motivating Example

In 2018, a firm in Australia lost 6.6 million dollars due to a backdoor in a smart contract. We will use the contract as an example to illustrate the threat of backdoor and the idea to detect this issue. This backdoor has been assigned with a CVE ID: CVE-2018-1000203 [8]. The backdoor function is listed as Listing 1.

```
function zero_fee_transaction(
    address _from,
    address _to,
    uint256 _amount
) onlycentralAccount returns(bool success) {
    if(balances[_from] >= _amount &&
        _amount > 0 &&
        balances[_to] + _amount >
        balances[_to]) {
            balances[_from] -= _amount;
            balances[_to] += _amount;
            Transfer(_from, _to, _amount);
            return true;
        } else{
            return false;
        }
}
```

Listing 1: A backdoor in SoarCoin contract, the "central Account" could transfer any token to any account, which leads to a loss of $6.6 million in 2018.

The *onlycentralAccount* in the header of the function is a modifier which asserts that the current account is the owner of the contract. The function changes the mapping structure *balances* to transfer some tokens from the address *_from* to the address *_to*. This function gives the owner a great power to get any token from any account, which is harmful to users' privacy. The "Transfer" in the function is an event trigger. An event is an interface defined by Solidity, which is used to write logs for the execution of EVM. Users used the keyword "event" to define a listener of an event. When the event was triggered, the backend of the system will catch

[1] We have submitted the 189 confirmed backdoors to NVD, 4 of them were verified by them and had been assigned with unique CVE identifiers, while others are still in the review process. The vulnerability and implementation: https://github.com/SmartContractBackdoor/Pied-Piper

it, and write the event into the log. "Transfer" here triggered an event defined by ERC20 [22]. The transfer function is ubiquitous in smart contracts. However, as defined by ERC20, a transfer process from an address to another address should get permissions of the *from* address. In this function, however, there is no approval verification process to permit the transfer, and the attacker had stolen 6.6 million dollars by exploiting this backdoor.

To detect the backdoor, Pied-Piper takes in the raw bytecode of the contract first, and decompiles it into an intermediate representation. Based on the representation, a control flow graph of the contract is built, in which each block contains some statements of opcodes, operands, and the results of the operations. Then a domain-specific Datalog analysis based on the CFG is designed. The analyzer detects the *onlycentralAccount* modifier first, and then checks whether it is a transfer-like function. Finally, Pied-Piper monitors the increment and decrement of elements in the mapping structures. If the rule that the path with token transfer should have approval statements is violated, there would be a potential backdoor. The workflow on this example is shown in Figure 1.

Figure 1: The work pipeline of Pied-Piper for this case. Pied-Piper checks three rules in this issue.

In the figure, the first red bar in the intermediate representation marks the statements that indicate an "onlycentralAccount" modifier. "CALLER" is an opcode used to push the address of the sender into the stack. The modifier uses an "EQ" opcode to assert the sender's address is the same as the owner's one. While in the second red bar, Pied-Piper identifies the transfer-like structure. The transfer-like function takes in three parameters. The first two parameters are addresses while the last one is an integer. Finally, the analyzer will check whether there is an approval statement along the path, and it will report a potential threat if no. In this way, Pied-Piper could successfully reveal this backdoor in a short time.

3 Background

3.1 Ethereum and Smart Contracts

The blockchain system is a distributed and shared platform based on cryptography, P2P transportation, game theory, and consensus mechanism. It consists of many linking blocks. Each block has a signature related to its last block, which ensures data integrity and data consistency. As a well-known blockchain platform, Ethereum leads in the concept of smart contracts. Smart contracts are programs deployed on Ethereum which are written in some high-level languages such as Solidity. Ethereum provides a virtual machine (EVM) to translate and execute smart contracts.

Smart contracts are deployed at some addresses on Ethereum, just like common accounts. The data used in smart contracts are stored at the contract's storage or the persistent storage of Ethereum. To prevent the waste of resources, Ethereum designs a gas mechanism. Gas is the necessary cost whenever a transaction takes place in Ethereum and need to be acquired by ether. In recent years, despite the rapid development of blockchain industry, there are more and more vulnerabilities exposed both in smart contracts and in EVM. Among these vulnerabilities, the smart contract backdoor receives increasing attention and exerts increasing influence.

3.2 Datalog Analysis

Datalog is a declarative programming language that was created in the late 70s. It is designed based on the syntax style of Prolog [46] and is widely used in program analysis. Datalog defines rules as well as facts to verify the logic of a program. Rules are boolean combinations of a series of atomic rules, and facts are abstracted logical representations based on the model built upon the program.

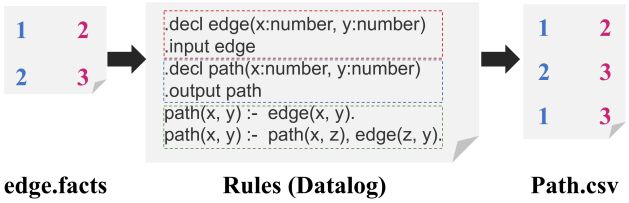

Figure 2: An example for Datalog analysis. The facts describe the edges between nodes and the *Rules* refer to the relations that a path needs to satisfy. The path.csv is the output result for this analysis process.

Figure 2 shows an example of facts and rules in Datalog analysis from the website of souffle tutorial [41]. This example analyzes whether there is a path between two nodes in a graph. As we can see in the figure, *edge.facts* describes the edge relations. In this example, there are two edges in the graph. The first edge is between node 1 and node 2 while the second one is between node 2 and node 3. Apart from facts, the rules file can be divided into three parts: 1) The first two lines declare the input format of the rules. In this example, this program reads in a file representing the two nodes that

generate this edge. 2) The next two lines declare the output format of the rules. Similar to the input format, the output of this case gives out a file each line of which consists of two numbers represent two nodes that have a path between them. 3) The last two lines declare the rules that a path needs to satisfy. In this case, if there is an edge between two nodes, there is a path between the nodes. If there is a path from node x to node z, and an edge between node z and node y, then there is a path from node x to node y. This is a transitivity relation. The output file is shown as the *Path.csv*, which indicates all paths in the graph according to the definitions in the rules.

4 Smart Contract Backdoor Study

In this section, we will give 5 common types of backdoors in smart contracts from the result of an empirical study. We have collected and read many relevant news about smart contract backdoors in the past three years [10], [38], [13], [37], [45], [47]. In addition, we also consulted many industrial programmers who are engaged in smart contract development and collected a large number of opinions about the definition of smart contract backdoors. After a comprehensive analysis, we summarize these five common types of backdoors[2], which could be exploited by two ways: First, a malicious owner of the contract or the user that deploy such honeypot contracts could exploit the backdoors to break the trading rules and meet their own profit. Second, an attacker who acquires the private key of the owner account may also abuse the backdoors to make a damage to the whole system of Dapp. For ease of understanding, we give the source code as the example though our work is based on the bytecode level.

4.1 Arbitrarily Transfer Threat

The first type of backdoor is *Arbitrarily Transfer*. This kind of threat allows the caller to transfer any token arbitrarily, that is, the caller could take away any token he likes from any address. **This backdoor is the main reason that causes a loss of 6.6 million dollars as we mentioned in Section 1.** Listing 1 shows the most typical real-world example of this type of backdoor. There are three key points in this problem. The first is an *onlycentralAccount* modifier. The second is a transfer-like structure. The structure requires three parameters, two of which are addresses. Some elements related to the first parameter are increased in the structure and other elements related to the second parameter are decreased. The third point is that there is no approval statement in this function. If a malicious owner exploits this backdoor, he could transfer tokens arbitrarily without approval, and all of the tokens in the Dapp belong to the attacker.

[2]In addition to these five kinds of smart contract backdoors, there are some other types of threats. However, they could only be exploited in extremely special circumstances, and is not within the scope of this work.

4.2 Generate Token Threat

The second type of the backdoor is *Generate Token*. This kind of threat allows the caller to generate tokens to any address after the ICO process. **Bancor contract was reported to have a backdoor (in the function named 'issue') in 2017 that could generate tokens arbitrarily at any time [45].** The value of the token is completely controlled by the contract owner. The code of this backdoor in Bancor contract [3] is listed in Listing 2. The function has only two parameters. The first is the address to which the generated token is given. The second is the amount of the tokens to be minted. The modifier *validAddress* is used to check whether the first parameter is a valid address. And the modifier *notThis* checks whether the first parameter is the same as the contract's address.

```
function issue(
    address _to,
    uint256 _amount
) public ownerOnly
 validAddress(_to) notThis(_to) {
    totalSupply =
    safeAdd(totalSupply, _amount);
    balanceOf[_to] =
    safeAdd(balanceOf[_to], _amount);
    Issuance(_amount);
    Transfer(this, _to, _amount);
}
```

Listing 2: An example for *Generate Token* backdoor, the owner account could generate tokens after ICO.

There are two key points of this issue. The first one is that there is an *ownerOnly* modifier. The second is that it is a transfer-like structure. However, this transfer-like structure takes in only two parameters. A token generating operation needs only one address variable to receive the tokens that are minted. This backdoor could generate any number of tokens, disrupt the market order and control the price of tokens somehow. A reasonable process for generating tokens may contain a modifier which asserts that it is in the process of ICO. If the modifier finds that the ICO process has finished, no more new tokens should be generated.

4.3 Destroy Token Threat

The third type of the backdoor is *Destroy Token*. **In the same report [45], the Bancor contract was also revealed with a backdoor that could destroy any token from any account at any time.** The wallet of each account is exposed to the contract owner. Listing 3 shows the code of this backdoor in function *destroy* of Bancor contract.

Similar to the *Generate Token* backdoor. There are also two key points for this backdoor vulnerability: *ownerOnly* modifier and a Transfer-like structure. Although some developers explain that this kind of function is used to destroy the tokens in some malicious accounts after they committed their attacks.

However, the team has the power to pick any account's tokens and destroy any amount of them, and this is a great threat to other users' privacy.

```
function destroy(
    address _from,
    uint256 _amount
) public ownerOnly{
    balanceOf[_from] =
    safeSub(balanceOf[_from], _amount);
    totalSupply =
    safeSub(totalSupply, _amount);
    Transfer(_from, this, _amount);
    Destruction(_amount);
}
```

Listing 3: An example for *Destroy Token* backdoor, the owner account could destroy tokens in any account.

4.4 Disable Transferring Threat

The fourth backdoor type is *Disable Transferring*. Some contracts have a function that could disable all the transferring operations. **In Bancor contract, there is also a backdoor that could stop all transfers reported in 2017 [45].** The team use this function to forbid transferring until their product is online. The tokens stored in users' accounts may be worthless without circulation. The code of this backdoor is shown as Listing 4.

```
modifier transfersAllowed {
    assert(transfersEnabled);
    _;
}
...
function disableTransfers(bool _disable)
    public ownerOnly {
    tranasferEnabled = !_disable;
}
...
function transfer(address _to, uint256 _value)
    public transfersAllowed
    returns (bool success){
    ...
}
```

Listing 4: An example for *Destroy Token* backdoor, the owner account could destroy tokens in any account.

The modifier *transfersAllowed* is used to check whether transferring is enabled for now. The backdoor function is *disableTransfers*. The owner could control the permissions of transferring by the variable *transfersEnabled*. Users could not commit any transfers due to this backdoor. All the tokens are forced to be locked by this function.

4.5 Freeze Account Threat

The last backdoor type is *Freeze Account*. **In 2019, a backdoor in SPAcoin's contract [43] that could freeze wallets (and addresses) was assigned with a CVE ID: CVE-2019-16944 [9]**[3]. This backdoor can destroy any assets of any accounts. There are totally 869 transactions based on this contract before November 13rd, which means the backdoor may have a wide influence on all the investors of this coin. The contract of the backdoor is deployed at address: 0x61402276c74c1def19818213dfab2fdd02361238 on Ethereum. Listing 5 shows the code of this contract.

```
function freezeAccount(
    address target,
    bool freeze
) onlyOwner public {
    frozenAccount[target] = freeze;
    FrozenFunds(target, freeze);
}
```

Listing 5: An example for *Destroy Token* backdoor, the owner account could destroy tokens in any account.

The function takes in two parameters, the first is an address that will be frozen or set free and the second is a bool variable that is used to control whether an account is frozen. *FrozenFunds* is an event trigger that emits a Frozen event. The account could not do anything due to this backdoor. Though this may be a mechanism to lock the bad accounts, it may also be harmful to normal users if the backdoor is abused.

5 Pied-Piper Design

In this section, we formally introduce the workflow of Pied-Piper. Figure 3 shows the working pipeline of Pied-Piper. There are two steps for Pied-Piper to identify a smart contract backdoor. The first step is to construct the facts based on the contract CFG for Datalog analysis. In this step, the bytecode is decompiled into an intermediate representation first, which is the CFG of the contract. Then, through a facts extraction process, Pied-Piper collects some basic structures and relations of the CFG. The second step is backdoor detection. Pied-Piper first defines some identifications of specific data structure related to backdoors. Then, based on these data structures, Pied-Piper identifies some function types, such as transfer function, freeze function, and so on. Finally, Pied-Piper detects backdoor threats on the basis of the function types as well as the data structures. The output of Pied-Piper is a report with types of backdoor threats.

The algorithm 1 shows the working principle of each step in Pied-Piper. Function *FactsConstruct* shows the details of the *Facts Construction* part in the figure. As shown in line 2 in the algorithm, a CFG is generated by the decompilation of the bytecode. Line 4 to line 8 in the algorithm describes the construction of structure facts. Structure facts represent the basic structure of a contract CFG, such as a block, an edge, and

[3]This vulnerability was detected and reported by Pied-Pier and was accepted in NVD with the unique CVE number

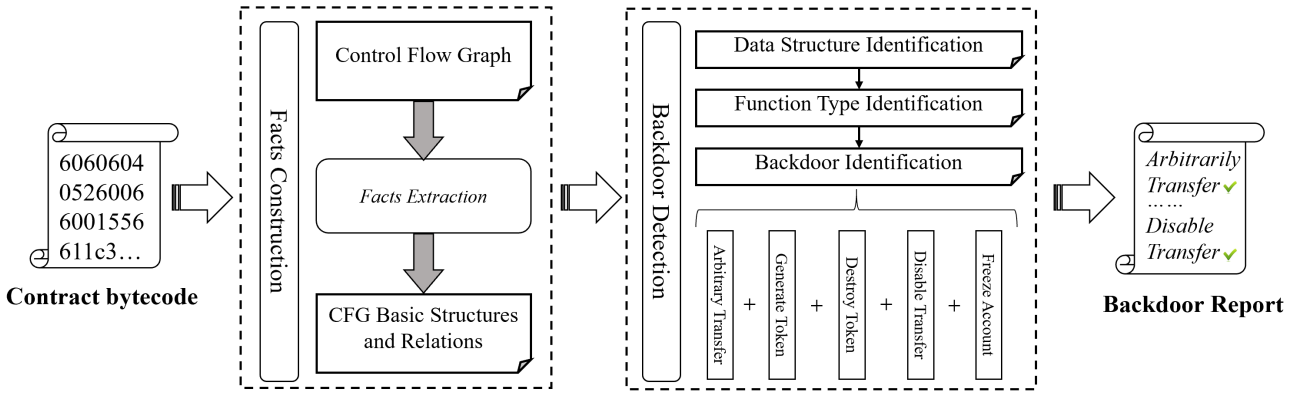

Figure 3: First, Pied-Piper decompiles the bytecode of contracts into an intermediate representation(CFG) for Facts construction. Through a facts extraction process, Pied-Piper collects some basic structures and basic relations of contract CFG as facts. Then, Pied-Piper hierarchically identifies five types of backdoors with Datalog analysis technique. Finally, Pied-Piper gives out a report.

Algorithm 1: Algorithms in Pied-Piper's pipeline

```
   Input: Hex: Raw bytecode of smart contracts
   Output: R: Reports with the types of backdoors in the contracts
1  Function FactsConstruct(Hex):
2      CFG = _decompile(Hex), i = 0, j = 0
3      facts=[]
4      while i < Structurefacts.length() do
5          newStructure = getFacts(CFG, Structurefacts[i])
6          facts.push(newStructure)
7          i++
8      end
9      while j < Relationfacts.length() do
10         newRelation = getFacts(CFG, Relationfacts[i])
11         facts.push(newRelation)
12         j++
13     end
14     return facts
15 End Function
16 Function Backdoor_Detection(facts):
17     Functions = getFunction(facts)
18     Reports = [], Report = null
19     for func in Functions do
20         DS = getDataStructure(func)
21         FunType = getFunctionType(func,DS)
22         Backdoors = getBackdoor(func,FunType,DS)
23         if Backdoors.length()!=0 then
24             Report = GenerateReport(Backdoors)
25             Reports.push(Report)
26         end
27     end
28     return Reports
29 End Function
30 Function Pied-Piper(Hex):
31     CFG = FactsConstruct(Hex)
32     R = Backdoor_Detection(CFG)
33     saveFile(R)
34 End Function
```

so on. Relation facts, however, describes some basic relations between the structures of a contract CFG. The process to construct relation facts are shown in line 9 to line 13 in the algorithm. The arrays *Structurefacts* and *Relationfacts* refer to all the types of facts defined in Pied-Piper. The details of facts construction are introduced in Section 5.1.

Function *Backdoor_Detection* takes in facts of the contract CFG, analyzes the constraints defined as domain-specific rules, and returns reports of the backdoor threats. As shown in line 17, based on the CFG, all the functions in the contract are identified first. As line 19 to line 27 describes, for each function of the contract, Pied-Piper will collect all the data structures as well as the function types. Then, Pied-Piper will check whether there are backdoor threats in the function. If there is any backdoor, shown at line 23 to line 26, a report will be updated. The details of the judgment of data structures and the function types will be introduced in Section 5.2.

5.1 Facts Construction

We build the facts of the smart contracts as the basic structures and relations of CFG. The definitions of basic structures are shown in Table 1.

As the table shows, Pied-Piper defines 5 types of basic structures. A *statement* is an operation consists of an opcode and its operands. A *block* is a sub-sequence of statements where no edges exist. Structure *Edge* in the figure means connectivity, not just the edges in the CFG. If one block is related to another block with a JUMP relationship, there is an edge between these blocks. Besides, the structure *Edge* is transitive, which means if there is an edge between block1

Table 1: Definitions about some basic structures in a smart contract CFG.

Name	Explanation
Statement(s)	s is a statement which represents the opcode as while as its operands in the opcodes sequence. (CFG)
Block(b)	b is a block consists of a series of statements. There is no edges in a block.
Edge(b1,b2)	If there is a JUMP relationship between block b1 and block b2, there is an edge. Besides, $edge(b1,b2) \cap edge(b2,b3) \rightarrow edge(b1,b3)$.
Variable(v)	v is a variable used or defined in a statement. That represents all the parameters and results in statements except constants.
Function(f)	f is a function defined in a contract, marked with a unique signature.

and block2, and an edge between block2 and block3, we also can find an edge between block1 and block3. A *variable* is a parameter or a result of a statement, opposite to the constant. A *function* is defined in smart contract code, which is marked with a unique signature.

Table 2: Some basic relations based on the basic structures defined in Table 1.

Notation	Explanation
op(op1:Opcode,s1: Statement)	A relationship between an opcode and a statement. s1 uses the opcode op1.
use(v1:Variable,s1: Statement, n:Number)	A use relationship refers that a statement use a variable v1 in the n-th position.
define(v1:Variable, s1:Statement)	The statement s1 defines a variable v1, that is, v1 is the result of the operation in s1.
stmtInfunc(s1:Statement, f1:Function)	A relationship between a statement and a function, indicates that statement s1 is used in function f1.
Value(v1:Variable, c:Constant)	The value of the variable v1 is constant c1.
stmtInblock(s1:Statement, b1:Block)	Statement s1 is used in block b1.

Based on these structures, Pied-Piper also defines some basic relations shown in Table 2. We use some new types in this figure: *Opcode* is a type used to represent an opcode defined by Ethereum. The type *Number*, however, represents an integer and type *constant* represents a constant value. There are also 5 basic relations defined by Pied-Piper. *op* is a relation that indicates an opcode *o1* is used in statement *s1*. And the *use* relation refers that a statement uses a variable *v1* in position *n*. The next relation *define* indicates that a statement defines a variable which means the variable is the result of the statement. The fourth basic relation is *stmtInfunc*. This relation indicates that a statement is in a function. *Value* relation means constant *c1* is the value of variable *v1*. The last basic relation is *stmtInblock* identifies that a statement is in a block.

5.2 Backdoor Detection

In this part, we will introduce the hierarchical rule definitions for backdoor detection.

5.2.1 Data Structure Identification Rule

Armed with the basic structures and the relations defined in the last section, Pied-Piper could define some data structure identification rules to identify backdoor related data structures in a contract function. As Figure 4 shows , all of the 5 rules are represented in the form of logic expressions. It needs to be mentioned that, all the different variables have different values in our definitions, which means, no variable's value is equal to another one in one expression. *Depends* is a relation that reveals the dependency of two variables. If a variable *v1* is defined in a statement uses another variable *v2*, we say that *v1* depends on *v2*. This relation also has transitivity, which means if *v1* depends on *v2* and *v2* depends on *v3*, then we could also say that *v1* depends on *v3*.

depends(***v1***:Variable, ***v2***:Variable) =>
$\{def(v1, statement) \cap use(v2, statement)) \cup (depends(v1, v3) \cap depends(v3, v2)\}$.

Parameter(***par***:Variable, ***stmt1***:Statement) =>
$\{\exists stmt1 \in S \mid op(stmt1, \text{"CALLDATALOAD"}) \cap def(par, stmt1)\}$.

AddressInParameter(***address***:Variable, ***stmt2***:Statement) =>
$\{\exists stmt1 \in S \mid op(stmt1, \text{"CALLDATALOAD"}) \cap def(var1, stmt1)\} \cap$
$\{\exists stmt2 \in S \mid op(stmt2, \text{"AND"}) \cap def(address, stmt2)\} \cap$
$(use(var1, stmt2,2))$.

MappingSub(***func***:Function, ***stmtSub***:Statement) =>
$\{\exists stmtMap \in S \mid op(stmtMap, \text{"SHA3"}) \cap def(from, stmtMap)\} \cap$
$\{\exists stmtSub \in S \mid op(stmtSub, \text{"SUB"}) \cap def(leftToken, stmtMap)\} \cap$
$\{\exists stmtLoad \in S \mid op(stmtLoad, \text{"SLOAD"}) \cap def(loadV, stmtLoad)\} \cap$
$\{\exists stmtStore \in S \mid op(stmtStore, \text{"SSTORE"})\} \cap$
$\{(use(from, stmtLoad, 1)) \cap (use(loadV, stmtSub, 1))$
$\cap (use(leftToken, stmtStore, 2))\} \cap$
$\{stmtInfunc(stmtSub, func)\}$.

MappingAdd(***func***:Function, ***stmtAdd***:Statement) =>
$\{\exists stmtMap \in S \mid op(stmtMap, \text{"SHA3"}) \cap def(from, stmtMap)\} \cap$
$\{\exists stmtLoad \in S \mid op(stmtLoad, \text{"SLOAD"}) \cap def(loadV, stmtLoad)\} \cap$
$\{\exists stmtAdd \in S \mid op(stmtAdd, \text{"ADD"}) \cap def(leftToken, stmtMap)\} \cap$
$\{\exists stmtStore \in S \mid op(stmtStore, \text{"SSTORE"})\} \cap$
$\{(use(from, stmtLoad, 1)) \cap (use(loadV, stmtAdd, 1))$
$\cap (use(leftToken, stmtStore, 2))\} \cap$
$\{stmtInfunc(stmtAdd, func)\}$.

Figure 4: Data structure identification rules defined with logic expression based on the basic structures and relations. S is the set of statements and the different variables in the expressions have different values.

The second relation *Parameter* identifies whether a variable is the parameter of a function, and return the statement that passes the parameter and the variable. We use an opcode named "CALLDATALOAD" to pass a parameter into a function. So we focus on the statement that uses this opcode and identify the parameter variable. Similar to *Parameter* relation, *AddressParameter* identifies an address parameter. The dif-

ference between address variables and other variables is that address variables need a transformation with "AND" opcode.

The last two relations are used to identify a subtraction operation and an addition operation on a mapping type in a transfer structure. Relation *MappingSub* identifies a function that has a subtraction operation of an element in a mapping structure. This relation first checks 4 statements. The first statement contains opcode "SHA3" and defines a variable named *from* in this expression. *SHA3* is an opcode defined by Ethereum to calculate a hash value of a given string. In this case, *SHA3* is used to calculate the storage address of the mapping elements. Besides, the statement that uses the opcode "SLOAD" is used to load the value from the storage. "SSTORE" is responsible for storing the result of the subtraction operation into the storage of the contract. After marking these statements, we should also give some more conditions on the variables used in these statements. Variable *from* should be used in the statement *stmtLoad* as the address of the loading operation. The loading variable should be one of the operands in the subtraction operation. The result of the operation should be used in *stmtStore* so that it could be stored in the storage. If all of the conditions hold, the functions that have the statements we mentioned above are identified by *MappingSub* structure. Structure *MappingAdd* is similar to *MappingSub* except for the subtraction operation.

5.2.2 Function Type Identification Rule

Based on these data structures, we can now define some backdoor related function types. The rules of these types are defined in Figure 5. There are 7 types of function defined by Pied-Piper. *Transfer* identifies a transfer-like structure in a function. A function is transfer-like if it satisfies such conditions: 1) It has two address parameters and an integer (could also be other types sometimes) parameter. In the expression, we use three variables:*to*, *from* and *amount* to represent three parameters of this function. 2) There is a subtraction operation as well as an addition operation on elements in a mapping structure (a mapping in a transfer function is always used to store the balance of each account.) So we use *MappingSub* and *MappingAdd* to check this condition.

The other two types *transferwithoutSub* and *transferwithoutAdd* are similar to *transfer*. *transferwithoutSub* is used to identify functions with token generated structures. It only has two parameters: an address variable and an integer. The address variable refers to the target of token generating. The integer represents the amount of token that will be generated. *transferwithoutAdd* is used to identify functions with token destroyed structures, which is similar to *transferwithoutSub*.

FrozeFunction identifies the function that used to freeze an account. This kind of function has two parameters. The first is the target that will be frozen, while the second is a bool variable controls the frozen state of the target account. In the expression shown in the figure, we first catch the result of

transfer(*func*:Function) =>
$\{AddressInParameter(from, stmt1)\} \cap \{AddressInParameter(to, stmt2)\} \cap \{Parameter(amount, stmt3)\} \cap \{MappingSub(func, stmtSub)\} \cap \{MappingAdd(func, stmtAdd)\}$.

transferwithoutSub(*func*:Function) =>
$\{AddressInParameter(to, stmt1)\} \cap \{!\,AddressInParameter(from, stmt2)\} \cap \{Parameter(amount, stmt3)\} \cap \{!\,MappingSub(func, stmtSub)\} \cap \{MappingAdd(func, stmtAdd)\}$.

transferwithoutAdd(*func*:Function) =>
$\{AddressInParameter(from, stmt1)\} \cap \{!\,AddressInParameter(to, stmt2)\} \cap \{Parameter(amount, stmt3)\} \cap \{MappingSub(func, stmtSub)\} \cap \{!\,MappingAdd(func, stmtAdd)\}$.

FrozeFunction(*func*:Function) =>
$\{AddressInParameter(target, stmt1)\} \cap \{Parameter(froze, stmt2)\} \cap \{\exists stmt0 \in S \mid op(stmt0, \text{"ISZERO"}) \cap def(frozeV, stmt0) \cap use(froze, stmt0, \_)\} \cap \{\exists stmtCmp \in S \mid op(stmtCmp, \text{"OR"}) \cap use(V1, \text{stmt}Cmp, 1) \cap use(V2, stmtCmp, 2)\} \cap \{depends(\text{V1}, frozeV)\} \cap \{depends(V2, target)\} \cap \{stmtInfunc(stmtCmp, func)\}$.

AllowTransfer(*func*:Function,*stmtLoad*:Statement) =>
$\{\exists stmtLoad \in S \mid op(stmtLoad, \text{"SLOAD"}) \cap def(allowV, stmtLoad)\} \cap \{\exists stmtJump \in S \mid op(stmtJump, \text{"JUMPI"}) \cap use(JumpV, \text{stmt}Jump, 1)\} \cap \{depends(JumpV, allowV)\} \cap \{stmtInfunc(stmtJump, func)\}$.

OnlyOwner(*func*:Function) =>
$\{\exists stmt \in S \mid op(stmt, \text{"CALLER"}) \cap stmtInblock(\text{stmt}, b1)\} \cap \{\exists stmt1 \in S \mid op(stmt1, \text{"EQ"}) \cap stmtInblock(stmt1, b1)\} \cap \{\exists stmt2 \in S \mid op(stmt2, \text{"SLOAD"}) \cap stmtInblock(stmt2, b1)\} \cap \{stmtInfunc(stmt, func)\}$.

approve(*func*:Function) =>
$\{\exists stmt \in S \mid op(stmt, \text{"MSTORE"})\} \cap \{MappingSub(func, stmtSub)\} \cap \{stmtInfunc(stmt, func)\}$.

Figure 5: Function type identification rules defined based on the data structures. Each type is part of a backdoor.

an opcode named "ISZERO". This opcode changes all the non-zero value into 0 and zero into 1. There are two sequent "ISZERO" that are used to make sure the value of the result has the same meaning of the original input. After a series of operations, the opcode "OR" is used to give the final result of the bool variable. The operands of the "OR" opcode depend on the parameters of the function.

AllowTransfer and *OnlyOwner* are used to identify modifiers in the function. *AllowTransfer* is a modifier that checks whether the transfer is allowed by the owner for now. The variable used to control this is named *allowV* in the expression, defined by a statement that uses "SLOAD" opcode. If the modifier doesn't hold, the transaction will revert. So there is a "JUMPI" opcode whose condition depends on the value of *allowV*. As for *OnlyOwner*, we identify three statements with "CALLER" opcode, "EQ" opcode, and "SLOAD" opcode. Modifier *OnlyOwner* asserts the caller of the transaction is exactly the owner's account. "CALLER" opcode is used to achieve the current caller's address. Then, the owner's address will be loaded with the help of opcode "SLOAD" from the storage. "EQ" opcode is used to commit the comparison process. To strength the constraints, these statements should be in the same block.

The last type is named *approve*, which is used to make the approval on transferring. Based on the rules of ERC20,

an address A could only get tokens from another address B through transferring operations with an approval by B. This function changes an element of a two-dimension mapping structure. The way to distinguish a two-dimension mapping structure from a single-dimension one is to identify the opcode "MSTORE". A two-dimension mapping uses memory for addressing while the single-dimension one only uses the storage. If a transfer function considers the approving process, it will use subtraction operation to decrease the approval tokens of the *from* address.

5.2.3 Backdoor Identification Rule

Since we have already defined several data structures as well as function types, we could try to define the backdoor identification rules. Figure 6 shows the rules used to detect the 5 types of backdoor threats.

ArbitraryTransfer(*func*:Function) =>
$\{transfer(func)\} \cap \{OnlyOwner(func)\} \cap \{!\, approve(func)\}$.

GenerateToken(*func*:Function) =>
$\{transferwithoutSub(func)\} \cap \{OnlyOwner(func)\}$.

DestroyToken(*func*:Function) =>
$\{transferwithoutAdd(func)\} \cap \{OnlyOwner(func)\}$.

FreezeAccount(*func*:Function) =>
$\{FrozeFunction(func)\} \cap \{OnlyOwner(func)\}$.

DisableTransfer(*funcAllow*:Function) =>
$\{transfer(func) \cup transferwithoutSub(func)$
$\cup\ transferwithoutAdd(func)\} \cap$
$\{AllowTransfer(func, stmtLoad)\} \cap$
$\{use(\text{Load}V, \text{stmt}Load, 1)\} \cap$
$\{OnlyOwner(funcAllow)\} \cap$
$\{op(stmtStore, \text{"}SSTORE\text{"}) \cap use(StoreV, stmtStore)$
$\cap\ stmtInfunc(stmtStore, funcAllow)\} \cap$
$\{Value(StoreV, v1) \cap Value(LoadV, v1)\}$.

Figure 6: Rules used to identify different kinds of backdoor threats based on the function types and data structures.

As the figure shows, there are 5 rules mapping with the 5 types of backdoors. To identify *Arbitrarily Transfer* threats, three conditions need to be satisfied. First, it is a function with an *OnlyOwner* modifier. Then, it is a transfer-like structure. Besides, no approving process has been done in the function, which means the transfer is not permitted by the paying account. To detect *Generate Token* and *Destroy Token*, we only need two conditions. First of all, an *OnlyOwner* modifier. Besides, there is a token generating structure (*transferwithoutSub*) or a token destroying structure (*transferwithouAdd*). As for *Freeze Account* backdoor, we should only find a *FrozeFunction* with an *OnlyOwner* modifier. *Disable Transferring* is a little more complex. There are two functions related to this kind of backdoor. The first one is a transferring function, and this could be a transfer-like structure, a token generating structure, or a token destroying structure. The other one is a function that used to change the value of the variable that could control the permission of the transfer process. We named this function *funcAllow* in the expression. This function should have an *OnlyOwner* modifier and there is a statement which contains an "SSTORE" opcode in this function. The transferring function, in the meantime, should have an *AllowTransfer* modifier and load a variable from the storage for the assertion. The variable which is stored in the *funcAllow* must have the same value with the variable which is loaded in the transferring function. In this way, Pied-Piper could detect all the common backdoors in smart contracts.

6 Implementation and Evaluation

To implement Pied-Piper, we draw support from some open-source programs. In general, we implement Pied-Piper based on Vandal [4]. Vandal is a static program analysis framework for Ethereum smart contract bytecode. Vandal decompiles the bytecode into an intermediate representation, based on which, we can build the control flow graph and define the basic facts. To implement the analysis rules for the five types of backdoors, we build the Datalog analysis program based on the backdoor-specific rules. Then, we make some experiments to answer two research questions (RQs):

RQ1 Is Pied-Piper accurate in detecting backdoors, i.e., any false positives or false negatives ?

RQ2 Is Pied-Piper efficient in detecting backdoors in real-world smart contracts ?

6.1 Dataset and Environment Setup

All experiments were performed atop a machine with 8 cores (Intel i7-7700HQ @3.6GHz), 16GB of memory, and Ubuntu 16.04.6 as the host operating system. We prepared two datasets for the evaluation.

- **Manual Created Dataset.** We prepared a dataset that consists of 100 smart contracts with certain types of backdoors. A backdoor is manually embedded in each contract with the help of professional smart contract developers. Each type is embedded into 20 smart contracts.
- **Real-World Smart Contracts.** We wrote a crawler script to download the raw bytecode of smart contracts from Etherscan [12], a browser for Ethereum and smart contracts. In total, we got 13484 real-world smart contracts, to evaluate the effectiveness of Pied-Piper on real backdoor detection.

6.2 Accuracy on Embedded Backdoors

We first analyze the accuracy of Pied-Piper in the 100 contracts with manually embedded backdoors. As shown in Fig-

ure 7, **Pied-Piper needs 8 seconds for each contract and reported 98 cases without false positives**. It failed in detecting 2 of the backdoors in this dataset, one for *Generate Token* backdoor and the other for *Destroy Token* backdoor.

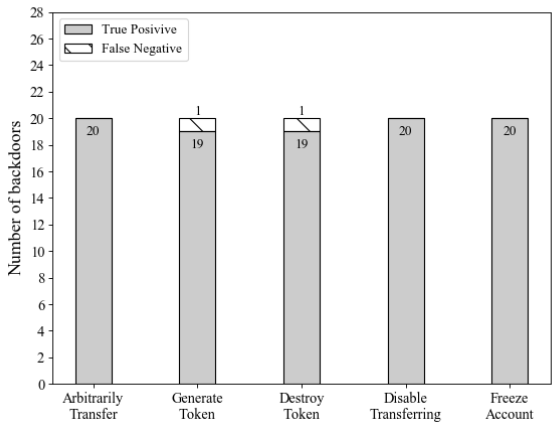

Figure 7: The distribution of the backdoor types in the manually labelled dataset, and the performance of Pied-Piper on each type. Pied-Pier successfully detects 98 threats out of the 100 embedded backdoors.

The first smart contract that results in an error of false negative in *Generate Token* backdoor detection is listed as Listing 6. This contract is deployed at address: 0xffdacbbf69b966ad6112daf996fce3e53bfdf97d. This contract is embedded with a *Generate Token* backdoor in function *MintToken*. The function, which contains an *onlyOwner* modifier, can generate any tokens to the owner's account.

```
// Code to mint tokens to the owner's account
function MintToken(
    uint256 amt
) public onlyOwner {
        require(balances[this] + amt >=
        balances[this]);
        currentSupply += amt;
        balances[this] += amt;
        emit Transfer(address(0), this, amt);
}
```

Listing 6: A token generating function with only one parameter which could not be detected by Pied-Piper.

However, during the datalog analysis, this function is not detected with the *transferwithoutSub* identification rule by violating the rule *AddreessInParameter*. This results in the failure of backdoor identification for *Generate Token*. The rule *AddressInParameter* expects for an address parameter while there is only one parameter whose type is *uint256* in the function *MintToken*. At first, we try to detect the increment of an element in a mapping structure without the limit to parameters. However, there are so many situations with mapping element increment which are not related to token generation. So we check whether there are exactly two parameters, one of which is an address variable and the other one is an integer. This rule definition could polish many false positives but results in the false negative of this example.

The second smart contract that results in a false negative in *Destroy Token* backdoor detection is shown as Listing 7. This contract is deployed at address: 0x07597255910a51509ca469568b048f2597e72504. The backdoor is embedded into the contract by two functions: *_burn* and *burn*. Tokens are destroyed in the internal function *_burn*. Function *burn* with an *onlyOwner* modifier is able to call the function *_burn* to exploit the backdoor.

```
// Code To Burn the token
function _burn(
    address account,
    uint256 value
) internal {
    require(account != address(0));
    totalSupply = totalSupply - value;
    balances[account] -= value;
    emit Transfer(account, address(0), value);
}
// Admin functionality to burn tokens
function burn(
    uint256 value
) onlyOwner public {
    _burn(msg.sender, value);
}
```

Listing 7: A token destroying function which is missed by Pied-Piper due to the violation of rule *Destroy Token*.

However, Pied-Piper finds that there is one function with an *onlyOwner* modifier, one function with *TransferWithoutAdd*, but no *Destroy Token* backdoors are detected. The main reason is that the *_burn* function lacks *onlyOwner* property. Since the *_burn* function is an internal function which can only be called by *burn* function in this contract, it doesn't need to be assigned with an additional attribute. In this way, Pied-Piper can only find one function with *onlyOwner* modifier and another function flagged as *TransferWithoutAdd* respectively. This kind of contract destroys the identification of rule *DestroyToken*, which requires these two constraints to appear in the same function. This results in a failure of detection on *Destroy Token* backdoor.

6.3 Efficiency on Real Smart Contracts

We evaluate Pied-Piper's efficiency on revealing backdoor threats in real-world smart contracts. On average, Pied-Piper needs 8.03 seconds (30.08 hours for all 13484 contracts) to make an analysis of a smart contract. 54.72% of the time is used in the decompilation process, and only 4.39

seconds are used in the analysis process. **In total, Pied-Piper reported 205 backdoors, with 189[4] confirmed and 16 false-positive errors identified by the smart contract developers.** Among the 189 confirmed backdoors, 4 of them have been assigned with unique CVE identifiers (CVE-2019-16944, CVE-2019-16945, CVE-2019-16946 and CVE-2019-16947) while others are still in the review process. The detail results are shown in Figure 8.

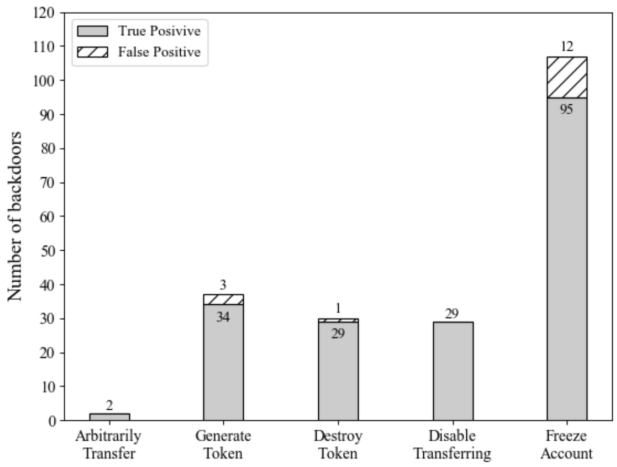

Figure 8: Backdoor threats revealed by Pied-Piper. Totally, there are 189 real backdoors found in all the 13484 contracts and 16 mislabeled samples.

From the result, we could see that backdoor *Freeze Account* is the most common type among these 5 types. The reason may be that many developers consider this type of backdoor as a protection mechanism when an accident happens. If someone steals the tokens or cheats in a transaction , this kind of function could be used as a reverting method to retrieve the loss. However, nobody could guarantee that this function will not be used in malicious situations. Besides, if the private key of the owner account is stolen [23], this kind of function may cause a disaster to all the users in this application. *Arbitrarily Transfer* is the least common type in these 5 types. However, this kind of backdoor may have the most serious impact (This backdoor has caused a loss of $6.6 million as described in Section 2). As for the other three types of backdoors, we can also tell from the figure that they are common in real contracts too. We will discuss more about those confirmed backdoors and false positives in detail below.

6.3.1 Real Backdoor Case Studies

In this section, we will give two real-world smart contracts as examples to illustrate how Pied-Piper detects the backdoor which could hardly be found without automatically datalog analysis. The first contract is deployed at address 0x0cb8d0b37c7487b11d57f1f33defa2b1d3cfccfe. The source code is listed in Listing 8.

The contract is special because there is neither *onlyOwner* modifier nor transfer-like structure in it. However, it is judged as a contract with a *Generate Token* backdoor by Pied-Piper. The reason for the judgment is that the contract calls a function in another contract named "MiniMeToken.sol". The function, named "generateTokens", is the same as listing 2. So Pied-Piper could also detect the backdoor threat in the contract which calls another function in other contracts. This backdoor could generate any amount of token to the owner's account, and anyone who has or steals the private key of the owner could exploit this backdoor to mint lots of tokens.

```
1  import "./MiniMeToken.sol";
2  contract DankToken is MiniMeToken {
3    function DankToken(
4    address _tokenFactory,
5    uint _mintedAmount
6  )MiniMeToken( _tokenFactory,0x0,
7      0,"Dank␣Token",18,
8      "DANK",true
9  ){
10     generateTokens(msg.sender, _mintedAmount);
11     changeController(0x0);
12   }
13 }
```

Listing 8: A contract with *Generate Token* backdoor. This contract calls a function (backdoor) in another contract.

The second backdoor is found in contract *ComBillAdvancedToken* deployed at 0x6292CEc07c345C6c6953e9166324f58db6D9F814. There is a *Freeze Account* backdoor in the contract, and this vulnerability has been assigned with a CVE identifier: CVE-2019-16947. The source code of the backdoor has been listed at Listing 9.

```
1  function approvedAccount(
2      address target,
3      bool freeze
4  ) onlyOwner public {
5      frozenAccount[target] = freeze;
6      FrozenFunds(target, freeze);
7  }
```

Listing 9: A contract with *Freeze Account* backdoor, this vulnerability has been assigned with a CVE identifier: CVE-2019-16947

The function *approvedAccount* can freeze any account at any time. The function has an *onlyOwner* modifier which satisfies the rule *onlyOwner* of Figure 5. Besides, as shown in line 5, the function changes the state of a storage object in the array through a boolean parameter. This satisfies the rule *FrozeFunction* of Figure 5. Then, Pied-Piper could successfully trigger the rule *FreezeAccount* of Figure 6, and detects this backdoor successfully. There are 3909 transactions made

[4]The contract with backdoors are listed at:https://github.com/SmartContractBackdoor/Pied-Piper

based on this contract before November 13rd, and this is really a serious threat for all the accounts of this application.

6.3.2 False-Positive Situations of Pied-Piper

As shown in Figure 8, we found 12 mislabeled samples in *Freeze Account* backdoor, 3 samples in *Generate Token* and 1 sample in *Destroy Token*. The reason leads to all of the false positives is semantic misunderstanding. That is, the functions satisfies all the identification rules of the backdoor but have a different usage. In this part, we will show typical false-positive cases and explain the reasons in detail.

```
1  function releaseLockedFund(
2      address _to,
3      uint256 _amount
4  ) public onlyOwner {
5      require(_to != 0x0,
6      "Valid␣Address␣Required!");
7      require(_amount <= lockedFund,
8      "Amount␣Exceeded␣Locked␣Fund");
9      lockedFund -= _amount;
10     balanceOf[_to] += _amount;
11 }
```

Listing 10: An incorrect Generate Token backdoor detected by Pied-Piper.

Taking the function presented in Listing 10 as an example, which is taken from the contract at: 0x4bd70556ae3f8a6ec6c4080a0c327b24325438f3. This function has an "onlyOwner" modifier, two parameters (an address and an integer), and a transfer-like structure (shown as line 10 in the code). The function satisfies the rule *GenerateToken* of Figure 6 during the Datalog analysis. However, the function is used to unlock some tokens to an account. The token transferred here is not generated arbitrarily. Pied-Piper could not distinguish this situation from the *Generate Token* backdoors for the misunderstanding of the semantics.

To give another example of semantic misunderstanding, we listed one of the contracts flagged with *Freeze Account* backdoor by Pied-Piper in Listing 11. It is part of the source code of a contract, named "MangoCoin". This contract is deployed at 0xc732713144eb34a4b6e802 032c9ccebf4f731036. The function *set_gamer* receives two parameters, one is an address type and the other is a boolean type. Besides, it changes the state of a storage object in the array through this boolean variable. So Pied-Piper regards the *set_gamer* function as a *FrozeFunction* type. With an assertion "msg.sender == st_owner" shown in line 6, the function also satisfies the rule *onlyOwner*. It satisfies all the constrains defined in rule *FreezeAccount*. However, the function is used to register a new gamer account. The boolean parameter here is not used to freeze certain account. Pied-Piper could not distinguish this situation from the *Freeze Account* backdoors for the misunderstanding of the semantics, which results in a false-positive alarm.

In summary, Pied-Piper can effectively detect real backdoors in Ethereum smart contacts with a relatively low false-positive rate and false-negative rate. On the performance of embedded backdoors detection, the false-positive rate is 0 and the false-negative rate is 2%. On the performance of 13484 real-world smart contracts, it detects 189 previous unknown backdoors with the false-positive rate of 7.8%. The analysis time for each smart contract is about 8.03 seconds in average. Furthermore, to avoid the impact of these backdoors, developers should standardize the development of smart contracts accordingly [17, 44] and control accounts with too much power.

```
1  function set_gamer(
2      address ad,
3      bool value
4  ) public {
5      require(ad != address(0x0));
6      require(msg.sender == st_owner ||
7      msg.sender == st_admin);
8      require(ad != st_admin);
9      newgamer[ad] = value;
10 }
```

Listing 11: An incorrect Freeze Account function detected by Pied-Piper.

6.4 Discussions

False positives and false negatives. As we illustrated above, Pied-Piper has some false positives because of the semantic understanding, as well as some false negatives because of the rules mismatch. Intuitively, almost all static analysis and dynamic analysis techniques have false positive and false negative problems. The static analysis suffers more from false positive problems due to over-fitting while dynamic analysis suffers more from false negative problems due to the incomplete scenario coverage for execution. Within our study, Pipe-Piper is with relatively very low false negative and false positive ratio, compared with the analyzer for other types of vulnerabilities of traditional software or smart contracts [16, 25, 31]. For false positives, one possible solution maybe executing the contract reported with a backdoor by dynamic analysis tool. During the execution, the semantic information can be extracted and analyzed to eliminate the false-positive situations. As for the false negatives, a possible solution maybe creating more detailed rules. More detailed rules mean a specific definition on the backdoor with one parameter and the backdoor that call other functions.

Manually Datasets. The first dataset used to evaluate the accuracy of Pied-Piper is embedded with backdoors manually. That means we may not contain all the possible situations and all possible backdoors. We consulted many developers of smart contracts to inject those backdoors, and this manual dataset is to our best-effort.

7 Related Work

Validation for Smart Contracts. Smart contracts have been shown to contain many vulnerabilities [2, 6, 19]. Once these defective codes are deployed on Ethereum, the attacker can easily exploit them to launch an attack, causing big loss. In 2016, hackers stole up to tens of millions of dollars from "The DAO" by making the best use of a re-entrant function flaw [32]. Due to the immutability of data, developers have limited abilities to patch the deployed contracts.

To ensure the security of Ethereum transactions, lots of research is devoted to detecting the security problems in smart contracts. These detectors can be divided into three categories: static analysis, dynamic analysis, and hybrid analysis. Zeus [25] and FSolidM [31] use abstract interpretation to verify the correctness and fairness of smart contracts. Static analysis can quickly analyze the code logic, but it also has a high false-positive rate. Authors in [26, 28, 33] apply symbolic execution to make up for this weakness, finding potential security bugs during execution. Recently, another dynamic method, like fuzzing, is applied to verifying the contract, like Reguard [27] and ContractFuzzer [24]. Echidna [7] generates random call sequences based on a list of invariants, for developers to check whether there is any problem.

Some works are aiming at protecting transactions on Ethereum on Ethereum Virtual Machine (EVM) level. KEVM [18] is an executable formal specification of the EVM's bytecode stack-based language built with the K Framework. This framework supports further formal analysis of the transaction process. EVMFuzz [14] uses differential fuzz testing to expose vulnerabilities in different EVM implementations. Another work EVM* [29] also tries to make instrumentation in the source code of EVM and detects dangerous operations during the execution process of transactions. Sereum [34] protects transactions from re-entrancy vulnerabilities.

Declarative Program Analysis. Using a declarative language for program analyzing will bring lots of benefits, which gives a way to express the developer's intent and leaves everything else to the underlying system. Nowadays, many declarative program analysis tools use a domain-specific-language (DSL), Datalog [21], for defining recursive relations. For example, the Doop framework [39] uses a logic-based language for points-to analysis of Java programs and unequivocally redefines the state-of-art in pointer analysis, proving that Datalog is useful and effective in program analysis. Zhang et al. [48] presents an effective method to find a relevant abstraction for program analyses written in Datalog , and authors in [30] also present Flix to specify and solve least fixed point problems. As a variant of Datalog engines, Souffle [42] is designed for tool designers crafting static analyses and provides the ability to rapid prototype.

For smart contracts, there are also some analysis tools based on Datalog. Vandal [4] is a static program analysis framework for smart contract, it converts the EVM bytecode into an equivalent intermediate representation, then feeds it to Datalog engine. Similarly, MadMax [16] uses Datalog analysis to explore out-of-gas vulnerabilities within smart contracts.

Backdoor Detection in Traditional Software. Backdoors are usually defined as some malicious codes hidden in a software system. Hackers usually exploit this piece of code to steal users' information or access protected resources [1, 15]. In [49], a special-purpose algorithm is represented to find backdoor codes based on keystroke characteristics. Horng et al. [20] use a two-layer detection system to identify keystroke interactive connection and record all dynamic link libraries. Another detection idea is to use clustering to distinguish system behavior and network traffic. With the help of Artificial Neural Network and genetic algorithm, this method can reach a high precision [36]. For deep learning systems, backdoors are more dangerous because they can threaten the lives of users and other people. When a deep learning model is trained on a dirty dataset, which has been inserted with crafted samples by adversaries, some malicious behaviors will happen during the execution. Chen et al. [5] uses the idea of activation clustering to verify and repair the ML model backdoors.

Main Difference. Different from the above work, we guarantee the security of smart contracts from another perspective and develop an efficient method to detect security risks caused by super authority. Our work uses the same inference engine as all Datalog-based work but leverages its advance for detecting hidden backdoors. Therefore, when any improvement or enhancement for the Datalog-based program analysis technique is proposed, our tool will also benefit from it.

8 Conclusion

In this work, we summarized 5 common types of backdoors in smart contracts based on the empirical study of existing reports and opinions given by the smart contract developers. There are generally two ways to exploit these backdoors, and would lead to a big loss of other users. The first is that some malicious owner of the contract may use the backdoors to meet their profit. The second is by acquiring the private key of the contract owner. Both of the methods break the rule of decentralization and may destroy the user's trust in the whole system. Besides, we designed and implemented Pied-Piper, a static analysis tool that could detect potential backdoor threats in smart contracts. Based on Datalog analysis Pied-Piper could successfully reveal all the five types of backdoors on the bytecode level. In the smart contracts deployed on Ethereum, Pied-Piper successfully found 189 confirmed backdoor threats.

Our future work will focus on two aspects. First, we will try to polish our Datalog program to reduce false negatives, and incorporate dynamic analysis into Pied-Piper to reduce false positives. Then, we will try to optimize the analysis process of Pied-Piper to make it more efficient.

References

[1] Narges Arastouie and M. R. Razzazi. Hunter: An anti spyware for windows operating system. *2008 3rd International Conference on Information and Communication Technologies: From Theory to Applications*, pages 1–5, 2008.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. In *IACR Cryptology ePrint Archive*, 2016.

[3] Bancor. Smarttoken. https://etherscan.io/address/0x1f573d6fb3f13d689ff844b4ce37794d79a7ff1c#code. Accessed November 4, 2019.

[4] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *ArXiv*, abs/1809.03981, 2018.

[5] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. *ArXiv*, abs/1811.03728, 2018.

[6] ConsenSys. smart-contract-best-practices. https://github.com/ConsenSys/smart-contract-best-practices/blob/master/docs/known_attacks.md. Accessed November 4, 2019.

[7] crytic. echidna. https://github.com/crytic/echidna/, 2019.

[8] CVE. Cve-2018-1000203. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000203. Accessed November 4, 2019.

[9] CVE. Cve-2019-16944. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16944. Accessed November 4, 2019.

[10] eric.eth. Just so everyone is aware, there is a backdoor in the usdc stablecoin launched by @coinbase today which allows any address to be blacklisted and funds frozen. https://twitter.com/econoar/status/1054785269843415040. Accessed November 4, 2019.

[11] Ethereum. Ethereum/solidity. https://github.com/ethereum/solidity. Accessed November 4, 2019.

[12] Etherscan. Etherscan. https://etherscan.io/. Accessed November 4, 2019.

[13] Hard Fork. Pax stablecoin has backdoor for freezing and seizing cryptocurrency. https://thenextweb.com/hardfork/2018/09/20/stablecoin-backdoor-law-enforcement/. Accessed November 4, 2019.

[14] Ying Fu, Meng Ren, Fuchen Ma, Yu Jiang, Heyuan Shi, and Jiaguang Sun. Evmfuzz: Differential fuzz testing of ethereum virtual machine. *arXiv preprint arXiv:1903.08483*, 2019.

[15] Jonatan Gómez and Dipankar Dasgupta. Evolving fuzzy classifiers for intrusion detection. 2002.

[16] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *PACMPL*, 2:116:1–116:27, 2018.

[17] GILAD HAIMOV. How to create an erc20 token the simple way. https://www.toptal.com/ethereum/create-erc20-token-tutorial. Accessed November 4, 2019.

[18] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018.

[19] Yoichi Hirai. Formal verification of deed contract in ethereum name service. *November-2016.[Online]. Available: https://yoichihirai. com/deed. pdf*, 2016.

[20] Shi Jinn Horng, Ming Yang Su, and J G Tsai. A dynamic backdoor detection system based on dynamic link libraries. 2008.

[21] Neil Immerman. Descriptive complexity. In *Graduate Texts in Computer Science*, 1999.

[22] Investopedia. What is erc-20 and what does it mean for ethereum. https://www.investopedia.com/news/what-erc20-and-what-does-it-mean-ethereum/. Accessed November 4, 2019.

[23] ise. Ethercombing: Finding secrets in popular places. https://www.ise.io/casestudies/ethercombing/. Accessed November 4, 2019.

[24] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. 2018.

[25] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, 2018.

[26] Ao Li and Fan Long. Detecting standard violation errors in smart contracts. *ArXiv*, abs/1812.07702, 2018.

[27] Chao Liu, Han Liu, Zhao Cao, Zhuotong Chen, Bangdao Chen, and A. W. Roscoe. Reguard: Finding reentrancy bugs in smart contracts. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, 2018.

[28] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. *IACR Cryptology ePrint Archive*, 2016:633, 2016.

[29] Fuchen Ma, Ying Fu, Meng Ren, Mingzhe Wang, Yu Jiang, Kaixiang Zhang, Huizhong Li, and Xiang Shi. Evm*: From offline detection to online reinforcement for ethereum virtual machine. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 554–558, 2019.

[30] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From datalog to flix: a declarative language for fixed points on lattices. In *PLDI*, 2016.

[31] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *Financial Cryptography*, 2017.

[32] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *J. Cases on Inf. Techn.*, 21:19–32, 2019.

[33] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC*, 2018.

[34] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. *ArXiv*, abs/1812.05934, 2018.

[35] Mauro Sacramento. Backdoor flaw sees australian firm lose $6.6 million in cryptocurrency. https://finance.yahoo.com/news/backdoor-flaw-sees-australian-firm-115323212.html. Accessed November 4, 2019.

[36] Elham Salimi and Narges Arastouie. Backdoor detection system using artificial neural network and genetic algorithm. *2011 International Conference on Computational and Information Sciences*, pages 817–820, 2011.

[37] ALLEN SCOTT. New research finds backdoor 'centralized control' in many icos. https://bitcoinist.com/icos-centralized-control-new-study/. Accessed November 4, 2019.

[38] Sead. This new stablecoin has a backdoor for freezing funds too. https://cryptonews.com/news/this-new-stablecoin-has-a-backdoor-for-\freezing-funds-too-2908.htm. Accessed November 12, 2018.

[39] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Datalog*, 2010.

[40] SoarLab. Soarcoin. https://etherscan.io/address/0xD65960FAcb8E4a2dFcb2C2212cb2e44a02e2a57E#code. Accessed November 4, 2019.

[41] Souffle. A simple example. https://souffle-lang.github.io/simple. Accessed November 4, 2019.

[42] Souffle. Souffle. https://souffle-lang.github.io/index.html. Accessed November 4, 2019.

[43] SPAcoin. Spacoin. https://etherscan.io/address/0x61402276c74c1def19818213dfab2fdd02361238. Accessed November 4, 2019.

[44] StackExchange. What is minting? how is minting prevented after ico? https://ethereum.stackexchange.com/questions/49867/what-is-minting-how-is-minting-prevented-af\ter-ico. Accessed November 4, 2019.

[45] Udi Wertheimer. Bancor unchained: All your token are belong to us. https://medium.com/unchained-reports/bancor-unchained-all-your-token-are-belong\-to-us-d6bb00871e86. Accessed November 4, 2019.

[46] WikiData. Prolog. https://www.wikidata.org/wiki/Q163468. Accessed November 4, 2019.

[47] CryptoGlobe Staff Writer. Coinbase's new stablecoin (usdc) could freeze funds and censor accounts. https://www.cryptoglobe.com/latest/2018/10/coinbases-new-stablecoin-usdc-could-freeze\-funds-and-censor-accounts/. Accessed November 4, 2019.

[48] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *PLDI*, 2014.

[49] Yin Zhang and Vern Paxson. Detecting backdoors. In *USENIX Security Symposium*, 2000.