Pied-Piper: Revealing the Backdoor Threats in Ethereum ERC Token Contracts

FUCHEN MA, MENG REN, LERONG OUYANG, and YUANLIANG CHEN,
Tsinghua University, China
JUAN ZHU, Hubei University of Arts and Science, China
TING CHEN, University of Electronic Science and Technology of China, China
YINGLI ZHENG and XIAO DAI, China Central Depository & Clearing Co., Ltd., China
YU JIANG and JIAGUANG SUN, Tsinghua University, China

With the development of decentralized networks, smart contracts, especially those for ERC tokens, are attracting more and more Dapp users to implement their applications. There are some functions in ERC token contracts that only a specific group of accounts could invoke. Among those functions, some even can influence other accounts or the whole system without prior notice or permission. These functions are referred to as contract backdoors. Once exploited by an attacker, they can cause property losses and harm users' privacy.

In this work, we propose Pied-Piper, a hybrid analysis method that integrates datalog analysis and directed fuzzing to detect backdoor threats in Ethereum ERC token contracts. First, datalog analysis is applied to abstract the data structures and identification rules related to the threats for preliminary static detection. Then, directed fuzzing is applied to eliminate false positives caused by the static analysis. We first evaluated Pied-Piper on 200 smart contracts, which are injected with different types of backdoors. It reported all problems without false positives, and none of the injected problems was missed. Then, we applied Pied-Piper on 13,484 real token contracts deployed on Ethereum. Pied-Piper reported 189 confirmed problems, four of which have been assigned unique CVE ids while others are still in the review process. Each contract takes 8.03 seconds for datalog analysis on average, and the fuzzing engine can eliminate the false positives within one minute.

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures;**

Additional Key Words and Phrases: Smart contract, backdoor detection, datalog analysis, directed fuzzing

ACM Reference format:
Fuchen Ma, Meng Ren, Lerong Ouyang, Yuanliang Chen, Juan Zhu, Ting Chen, Yingli Zheng, Xiao Dai, Yu Jiang, and Jiaguang Sun. 2023. Pied-Piper: Revealing the Backdoor Threats in Ethereum ERC Token Contracts. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 61 (April 2023), 24 pages.
https://doi.org/10.1145/3560264

Authors' addresses: F. Ma, M. Ren, L. Ouyang, Y. Chen, Y. Jiang, and J. Sun, Tsinghua University, Beijing, 100000, China; emails: mafc19@mails.tsinghua.edu.cn, rm19@mails.tsinghua.edu.cn, lerong@connect.hku.hk, sard.chen@gmail.com, jiangyu198964@126.com, sunjiaguang@mails.tsinghua.edu.cn; J. Zhu (corresponding author), Hubei University of Arts and Science in Long zhong Road in Xiang Yang, Hubei, China, 441021; email: journey1022@126.com; T. Chen, University of Electronic Science and Technology of China, on Jianshe Road in Chengdu, China 610056; email: brokendragon@uestc.edu.cn; Y. Zheng and X. Dai, China Central Depository & Clearing Co., Ltd. on Jinrong Road in Beijing, China, 100000; emails: zhengyl@chinabond.com.cn, daixiao@chinabond.com.cn.

1049-331X/2023/04-ART61 $15.00
https://doi.org/10.1145/3560264

1 INTRODUCTION

Ethereum is a decentralized platform that supports smart contracts. Users can develop smart contracts in a high-level language such as Solidity [12] and deploy the contracts on the platform. The source code of smart contracts will be compiled to low-level bytecode and then executed by **Ethereum Virtual Machine (EVM)**. Since its creation, Ethereum has attracted more and more users. There are approximately 11.2 transactions [13] per second nowadays on Ethereum, and most of the transactions are financially related. So, it is 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 endlessly. Among all the contract vulnerabilities, threats related to high-privileged functions in ERC token contracts are often overlooked and thus create significant potential risks for users' property and privacy. This type of threat is defined as backdoors. **In June 2018, one firm in Australia lost $6.6 million due to a backdoor function in SoarCoin contract [40, 44].** This case raised widespread concerns, and the public thus began to focus on the threat caused by these special functions. However, it is difficult for users to judge whether there is such a threat in the contract when directly examining the code without professional knowledge. There are many works aimed at detecting software backdoors. However, the previous work could not be adapted to this problem because of the different definitions of the backdoor. Specifically, backdoors in traditional software refer to the way to cheat the permission authentication process. While in deep learning systems, a backdoor always means a poisoned dataset for malicious usage. Smart contract backdoors are even harder to detect because it is challenging to distinguish backdoor functions from the normal high-privilege functions. Meanwhile, it is hard to recognize source code structures on the bytecode level, which is necessary for smart contract testing because most of the contracts on Ethereum have no source code available.

In this work, we propose Pied-Piper, a hybrid analysis method that can automatically detect the potential backdoor threats in Ethereum ERC token contracts. First, we analyze and demonstrate five common types of backdoor problems with a detailed empirical study of many real contracts. The first type is *Arbitrary Transfer*, which permits the malicious attackers to transfer any amount of tokens from any address to another. The second type is *Generate Token After ICO*, the owner can generate any amount of tokens even after the ICO process has finished. The third one is *Destroy Token*, which refers to destroying any amount of tokens from some specific addresses. 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.

Pied-Piper used domain-specific datalog analysis and directed fuzzing to identify those threats. The datalog analysis engine builds the contract's **control flow graph (CFG)** to abstract the threats related data structures and identification rules. It analyzes the CFG to check whether the constraints described in the rules are violated or not for preliminary static detection. The fuzzing engine is designed to eliminate the false positives caused by the datalog analysis. The potential risks reported by the datalog analysis will be set as targets for fuzzing detection. After deploying the contract on a local chain, Pied-Piper dynamically executes the contract functions and saves the seeds that are closer to the target. If the fuzzing tool can touch the target statements and trigger a protection and interruption mechanism, the reported function is not a real threat, and the false positive could be eliminated precisely. This way, Pied-Piper can detect the contract backdoor problems.

For evaluation, we implemented Pied-Piper based on Vandal [4], an analysis framework for extracting program properties. We first tested Pied-Piper on 200 contracts manually injected with different backdoor threats. It reported all threats without false positives, and none of the injected

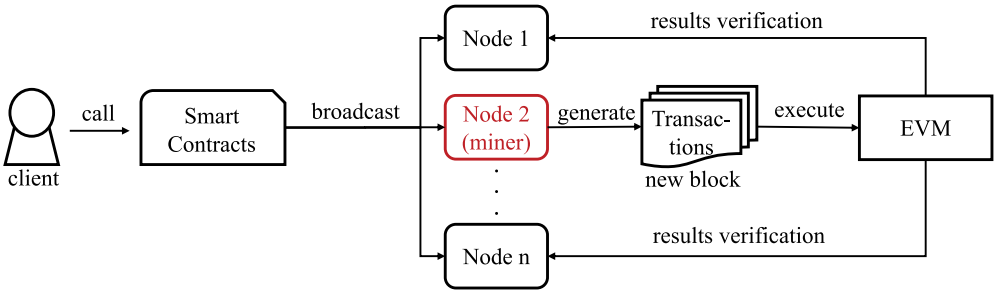

Fig. 1. The workflow of the transaction processing in Ethereum. A client calls a contract and emits a transaction. Then the miner generates a new block containing the transaction and executes it with EVM. Afterwards, the miner broadcasts the transaction results to other nodes for verification.

problems was missed. **Then, we applied Pied-Piper on 13484 real-world ERC token contracts and found 189 confirmed threats,[1] which the contract developers had confirmed.** Specifically, two contracts have been found with *Arbitray Transfer*, 34 with *Generate Token After ICO*, 29 with *Destroy Token*, 29 with *Disable Transferring*, and 95 with *Freeze Account*. It took around 30 hours to analyze these contracts for the datalog analysis engine. Each contract took an average of 8.03s. Three contracts reported with *Generate Token After ICO* problem during the datalog analysis have been proved with no threats by fuzzing engine within about one minute of the fuzzing process. The results show that Pied-Piper is effective and efficient in revealing backdoor problems in real-world smart contracts. Overall, our work makes the following contributions:

- We systematically investigated the five common types of backdoor problems in ERC token contracts. To our knowledge, we are the first to formulate the backdoors in smart contracts with a detailed empirical study.
- We designed and proposed Pied-Piper, the first hybrid analysis tool that could automatically analyze whether a smart contract has a backdoor threat.
- We implemented Pied-Piper and conducted several experiments to show the effectiveness of Pied-Piper. With Pied-Piper, we have found 189 confirmed threats in 13,484 real-world smart contracts. Four of them are assigned with CVE identifiers.

2 BACKGROUND OF ETHEREUM AND SMART CONTRACTS

Ethereum is considered as the second largest blockchain platform in the world [11], right after the Bitcoin system [3]. Powered by the consensus mechanism POW and POS, Ethereum coordinates all the nodes to make agreements on the transaction results. Smart contracts are the programs written in Solidity running on Ethereum. Based on smart contracts, developers have created various decentralized applications including games, decentralized finance, and so on. Ethereum leverages a virtual machine to translate the bytecode of the smart contracts and execute the operations. Such virtual machines, known as the EVM, are equipped in each Ethereum node.

Figure 1 shows the workflow of the transaction processing in the Ethereum system. When a client emits a transaction based on a smart contract, it will first broadcast the transaction to the transaction pool of all nodes. Based on the consensus algorithm POW or POS, Ethereum will then select one of the nodes as the miner who will generate the next new block. The miner then selects

[1]We have submitted all the confirmed problems to NVD, four of them were verified by them and had been assigned with unique CVE identifiers, while others are still in the review process. The vulnerabilities and implementation: https://github.com/EthereumContractBackdoor/PiedPiperBackdoor.

```
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. An arbitrary transfer problem in SoarCoin contract, the "central Account" could transfer any token to any account, which led to a loss of $6.6 million in 2018.

certain transactions from the transaction pool and packs them into the new block. Afterwards, the miner executes the transactions one by one with EVM. After the execution, it will send the results to all other nodes for verification. If the verification succeeds, all the transactions in the block will be committed. Otherwise, if the miner faked the execution results and the verification fails, the transactions will be put into the pool again and pending for processing.

3 MOTIVATING EXAMPLE

In 2018, a firm in Australia lost 6.6 million dollars due to an arbitrary transfer problem in a smart contract. The owner (Soar Labs) of the contract has claimed the existence of the backdoor [40]. This case has been treated as a criminal investigation by law enforcement. As a result, Soar Labs paid a compensation of $1.7 million and 5 million Soarcoins, and they also gave back all the shares in the Australian firm they acquired. We will use this contract as an example to illustrate the threat of arbitrary transfer problem and our idea to detect it. This event has been assigned with a CVE ID: CVE-2018-1000203 [8]. The involved function is listed in Listing 1.

The *onlycentralAccount* in the function's header is a modifier that asserts that only the owner of this contract can call it. The mapping structure *balances* is used to transfer some tokens from the address *_from* to the address *_to*. This function gives the owner the privilege to take or get any token from any account, which harms 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 EVM execution. Users could use the keyword "event" to define a listener of an event. When the event is triggered, the backend of the system will catch it and write the event into the log. "Transfer" here triggered an event defined by ERC20 [21]. As defined by ERC20, a transfer process from an address to another address should get permissions of the *from* address. However, there is no approval verification process in this function to permit the transfer operation, and the attacker had stolen 6.6 million dollars by exploiting this function.

To detect this threat, Pied-Piper takes in the source code of this contract and builds the control flow graph. Then a domain-specific Datalog analysis based on the CFG is designed. The analyzer detects the *onlycentralAccount* modifier first. The modifier uses an "EQ" opcode to assert that the sender's address is the same as the owner's. Then the datalog engine checks whether it is a transfer-like function. A transfer-like function requires three parameters. The first two parameters are addresses or arrays of addresses, while the last one is an integer. 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 problem.

4 SMART CONTRACT BACKDOOR STUDY

In this section, we will give five common types of backdoors in smart contracts from the result of an empirical study. We have collected and read more than 50 relevant news about ERC token contract backdoors in the recent years[2,3], such as [10], [42], [15], [41], [50], and [51]. In addition, we consulted many industrial programmers engaged in smart contract development and collected many opinions about the definition of ERC token contract backdoors. Specifically, we contacted 10 smart contract and blockchain developers during this study. We collected and analyzed these blogs and reports by checking the source code of the corresponding smart contracts with backdoor threats. Then we distributed our findings to the developers. The final list of threats is defined by merging all the opinions from the developers. After a comprehensive analysis, we summarize these five common types of backdoors.[4] They could be exploited in two ways: First, a malicious contract owner or the user that deploys such honeypot contracts could exploit the backdoors to break the trading rules and meet their profit. Second, an attacker who acquires the private key of the owner account may also abuse the backdoors to damage the 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. The caller could take away any token he likes from any address. **This backdoor is the main reason that caused a loss of 6.6 million dollars as we mentioned in Section 1.** Listing 1 shows a 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 others 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.

4.2 Generate Token After ICO Threat

The second type of the backdoor is *Generate Token After ICO*. This 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 [50].** The value of the token is entirely controlled by the contract owner. The code of this backdoor in Bancor contract [2] 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. Moreover, the modifier *notThis* checks whether the first parameter is the same as the contract's address.

There are two critical 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

[2] https://www.cybavo.com/blog/how-self-deployment-of-erc-20-smart-contracts-can-enhance-transaction-security/.
[3] Penny Wise and Pound Foolish: Quantifying the Risk of Unlimited Approval of ERC20 Tokens on Ethereum.
[4] In addition to these five kinds of smart contract backdoors, there are some other types of threats. However, they could only be exploited in exceptional circumstances and are not within this work's scope.

```
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 After ICO* backdoor, the owner account could generate tokens after ICO.

only two parameters. A token generating operation needs only one address variable to receive the minted tokens. This backdoor could generate any number of tokens, disrupt the market order and somehow control the price of tokens. A reasonable process for generating tokens may contain a modifier that 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 [50], 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 After ICO* backdoor. There are also two critical points for this backdoor vulnerability: *ownerOnly* modifier and a Transfer-like structure. However, some developers explain that this kind of function is used to destroy the tokens in some malicious accounts after committing their attacks. However, the team has the power to pick up any account's tokens and destroy any amount of them, and this is a significant 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 the Bancor contract, there is also a backdoor that could stop all transfers reported in 2017 [50].** 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.

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.

```
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.

4.5 Freeze Account Threat

The last backdoor type is *Freeze Account*. **In 2019, a backdoor in SPAcoin's contract [46] that could freeze wallets (and addresses) was assigned with a CVE ID: CVE-2019-16944 [9].**[5] This backdoor can destroy any assets of any accounts. There are 869 transactions based on this contract, which means the backdoor may have a vast 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 malicious accounts, it may also harm normal users if the backdoor is abused.

4.6 Avoid the Affects of Backdoors

To avoid the influences caused by backdoors, we summarized some advice for both Dapp users and smart contract developers.

For the Dapp users: We suggest the dapp users pay attention to the transfer, minting or destroying functions of the smart contracts corresponding with the Dapp. If these functions can be called by only a specific group of accounts and may have an influence on the other accounts' balance, it may be leveraged to cause a huge loss. Users should be careful to put their digital assets to Dapps with such functions.

[5]This vulnerability was detected and reported by Pied-Pier and was accepted in NVD with a unique CVE number.

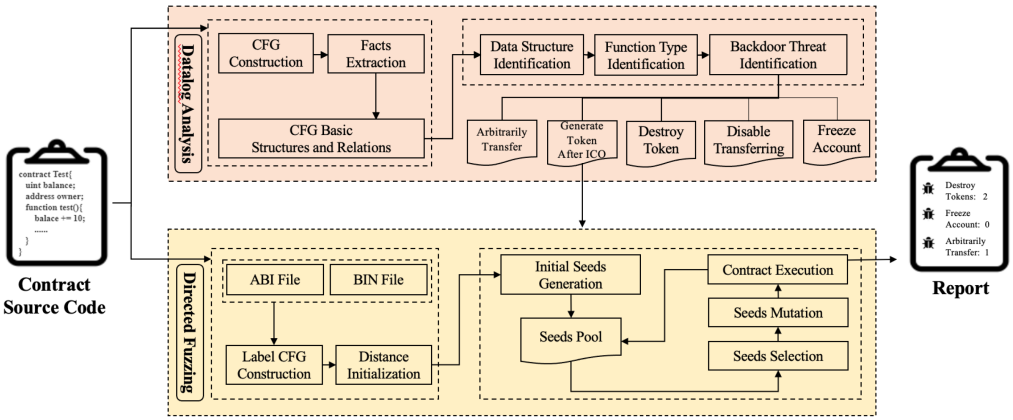

Fig. 2. First, datalog analysis is applied to abstract the data structures and identification rules related to the threats for preliminary static detection. Then, Pied-Piper applies directed fuzzing to make up for the unsoundness caused by the static analysis, especially for the elimination of false positives.

For the smart contract developers: Backdoor threats may affect the trustworthiness of the Dapp and if leveraged by malicious developers, they will damage the ecosystem of your applications. Thus, during the smart contract development process, it is essential to avoid such threats. According to our findings, developers have some ways to avoid backdoor threats:

(1) Arbitrarily Transfer Threat: The transfer of the tokens should always be approved by the 'from' address. This can be accomplished by the approve function described in ERC-20's document [37].
(2) Generate Tokens After ICO: A smart contract should not mint new tokens after ICO. If the business logic indicates that it is necessary to mint new tokens after ICO, a more convinced way is to add a voting mechanism that requires all the accounts to vote for the mint decision.
(3) Destroy Tokens: Like the token generating threats, a smart contract should not destroy tokens directly decided by the owner. If the token destroying logic is necessary (for example, destroy tokens of a malicious account), add a voting mechanism that requires all the accounts to vote for the destroy decision.
(4) Disable Transferring: Similarly, adding a voting mechanism for the transferring disabling decision for certain accounts is more convinced.
(5) Freeze Account: Add a voting mechanism for the account freezing and unfreezing decision rather than freeze the account directly by the owner.

5 PIED-PIPER DESIGN

In this section, we formally introduce the workflow of Pied-Piper. As presented in Figure 2, there are two steps to identify a backdoor threat. The first step is to make a static datalog analysis of the source code. In this step, Pied-Piper will first construct a CFG based on the contract's source code and collect some basic data structures and relations of the CFG. Then, Pied-Piper defines some identifications of specific data structures related to backdoor functions. Pied-Piper identifies some function types, such as transfer and approves functions based on these data structures. Finally, Pied-Piper detects a backdoor risk based on well-defined rules. The datalog analysis will give a preliminary report on the three types of backdoor problems. However, the static analysis of *Transfer In Tokens* type is not sound, and Pied-Piper uses a fuzzing engine to eliminate the false positives. The fuzzing engine will compile the contract and construct a new CFG with target label and

Table 1. Definitions About Some Basic Structures in a Smart Contract CFG

Name	Explanation
Statement(s)	s is a statement which represents the opcode while as its operands in the opcodes sequence.
Block(b)	b is a block consists of a series of statements, which starts with a jump target and ends with a JUMP or JUMPI opcode.
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.

node distance according to the location of the potential threats reported by the datalog analysis. If the guided fuzzing engine can reach the target statements and trigger a protection mechanism, the reported function is not a real threat, and the false positive could be eliminated precisely.

5.1 Datalog Analysis Engine

In the datalog analysis engine, we first 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 five types of basic structures. A *statement* is an operation consisting of an opcode and its operands. A *block* is a sequence of statements that starts with a jump target and ends with a JUMP or JUMPI opcode. 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 and block2 and an edge between block2 and block3, we can also say that there is 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.

Based on these structures, Pied-Piper also defines some fundamental 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. Type *Number* represents an integer and type *constant* represents a constant value. There are also six 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 *stmtInblock* relation represents that a statement is in a block. Similarly, the last relation *inFunction* identifies that a block is in a function.

5.1.1 Data Structure Identification Rule. Based on the basic structures and the relations defined in the last section, Pied-Piper could define some data structure identification rules to identify backdoor problems related data structures in a contract function. Figure 3 shows all five rules represented in the form of logic expressions. It must be mentioned that all the different variables have

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 to that a statement uses 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.
inFunction (b1: Block, f1: Function)	Block b1 is in function f1.

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

Fig. 3. Data structure identification rules defined with logic expression based on the basic structures and relations. 'S' in the rules represent the set of all statements in the smart contract's datalog and the different variables in the expressions have different values.

different values in our definitions, meaning 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 *v1* depends on *v3*.

The second relation *Parameter* identifies whether a variable is the parameter of a function and returns 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 identifies the parameter variable. Similar to *Parameter* relation, *AddressParameter* identifies an address parameter. The difference between address variables and other variables is that address variables need a transformation with the '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 with a subtraction operation of an element in a mapping structure. This relation first checks four 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 in the contract's storage. 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 operation result should be used in *stmtStore*, so it can 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.

5.1.2 Function Type Identification Rule. We can now define some backdoor problems related to function types based on these data structures. The rules of these types are defined in Figure 4. There are seven types and a particular relation between two functions 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 and 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 is used to freeze an account. This kind of function has two parameters. The first is the frozen target, while the second is a bool variable that controls the frozen state of the target account. In the expression shown in the figure, we first catch the result of an opcode named 'ISZERO'. This opcode changes all the non-zero values into 0 and zeroes into 1. Two sequent 'ISZERO' are used to ensure the value of the result is the same as 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 function's parameters.

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 does not 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 that the caller

Fig. 4. Function type identification rules defined based on the data structures. 'B' in the rules means the set of all the blocks. Each type is a part of a backdoor problem or related to backdoor operations.

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 strengthen the constraints, these statements should be in the same block.

The second last type is *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 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 storage. If a transfer function has an approving process, it will use the subtraction operation to decrease the approval tokens from the *from* address.

In some cases, different components may be located in different functions, such as a calling from a function with *OnlyOwner* modifier to an internal function containing a transfer structure. In order to detect backdoor problems in this situation, we designed a *call* relationship between two different functions. When there exists a block inside each function, and there is an edge connection between them in CFG, indicating a jump relationship between the two blocks, it is proved that there is a calling relationship between these two functions.

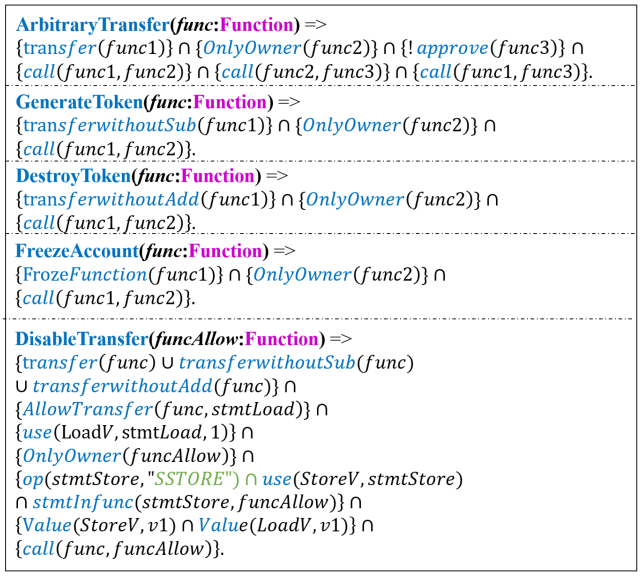

Fig. 5. Rules to identify different kinds of backdoor threats based on the function types and data structures.

5.1.3 Backdoor Identification Rule. Since we have already defined several data structures and function types, we could try to define the backdoor identification rules. Figure 5 shows the rules mapping with the five manifestations of backdoor problems.

Three conditions need to be satisfied to identify *Arbitrarily Transfer* threats. 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 paying account does not permit the transfer. We only need two conditions to detect *Generate Token After ICO* and *Destroy Token*. First of all, an *OnlyOwner* modifier. Besides, there is a token generating structure (*transferwithoutSub*) or a token destroying structure (*transferwithouAdd*). If the functions that each component is located in are different, there needs to be a calling relationship between these functions. However, for the detection of *Generate Token After ICO*, the datalog analysis engine of Pied-Piper is unsound because it cannot distinguish *Generate Token After ICO* from the normal token minting function in an ICO process. The importation of the rule to judge ICO stopping may lead to more false positives. Pied-Pier relays on dynamic analysis to eliminate the unsoundness caused by this situation.

```
1 // set the owner in construct function
2 function constructor(){
3     owner = msg.sender();
4     ...
5 }
```

Listing 6. The construct function of the contract sets the caller of the deployment transaction as the owner.

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, which could be a transfer-like structure, a token generating structure, or a token destroying structure. The other one is a function that is 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 that 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 stored in the *funcAllow* must have the same value as the variable loaded in the transferring function.

5.2 Directed Fuzzing Engine

In order to make up for the unsoundness of the static datalog analysis, especially for the elimination of false positives of the *Generate Token After ICO* threats, Pied-Piper deploys the contract on a local environment and executes the target function with the customization of directed fuzzing technique [5, 26, 49]. Generally, an ICO process of an ERC-20 token usually sets a fixed amount of supply tokens first. The ICO process will stop if all of the supply tokens have been distributed. Motivated by this, Pied-Piper marks all the modifier statements as targets and uses a fuzzing engine to execute each contract function. If the target is executed and a safe mode of an ICO process is triggered, Pied-Piper will stop the current fuzzing process and eliminate the threat as a false positive. The corresponding threat is reported as a true positive if the target is executed without triggering an ICO safe mode.

As Figure 2 shows, the fuzzing engine first preprocesses the smart contract source code. The preprocessing of the contract tries to delete the *onlyOwner* modifier. In most cases, the owner of a contract will be set as the caller of the deployment transaction of the contract as shown in Listing 6. In this case, the fuzzing engine can precisely set the node in the local chain as the contract's owner and ignore the effects of *onlyOwner* modifier.

However, there are also some cases where the contract owner is set as a fixed address. As shown in Listing 7, the owner's address of the contract is a constant, and the fuzzing engine cannot set any node in the local chain as the owner. Thus, before fuzzing starts, we consider deleting all of the *onlyOwner* modifiers in the contracts. Pied-Piper will take in the suspect function reported by the datalog analysis engine. The modifier-related statements in the suspect function will be set as the fuzzing target in the fuzzing process. Then, Pied-Piper will generate a new CFG based on the processed contract, flag the nodes related to the targets, and compile the contract into an abi file and a bin file for fuzzing. An abi file, whose full name is the application binary interface, describes the functions, events, and some related information of a contract. With the help of an abi file, we can extract the parameter information of each function in the contract.

As Figure 2 shows, there are mainly three steps in the fuzzing process. Firstly, Pied-Piper will generate an initial seed for the first execution. The seed has two features: the order of the functions and the input of the functions. At the very beginning, the order of the function is the same as the order in the abi file. The inputs are generated randomly according to the type of each parameter of the functions. However, as we illustrated at the beginning of this section, ICO process always sets a fixed amount of total supply tokens. In order to reach the value of total supply tokens in the fastest way, Pied-Piper always sets the maximum value of each numeric type variable, such as uint, uint256, etc.

Armed with the initial seed and the binary code of the contract, Pied-Piper can execute the suspected function given by the datalog analysis engine. After each round of the execution, Pied-Piper mutates the seed by disordering the function and changing the inputs randomly. Each function will be executed again under the new mutated seeds. In order to select good seeds for the next round of execution, we directly reserve the seeds which have a shorter distance to the target nodes in a CFG. A distance between two nodes in a CFG is defined in Formula 1:

$$D_n(n, T_n) = \begin{cases} [\sum_{t_n \in T_n} (\frac{1}{d(n, t_n)})]^{-1}, & n \notin T_n. \\ 0, & n \in T_n. \end{cases} \quad (1)$$

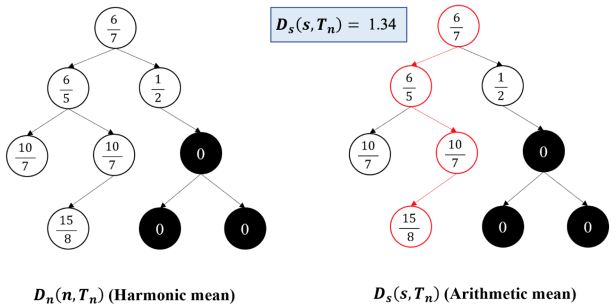

Fig. 6. Distance between a node and target nodes and the distance between a seed trace and target nodes.

```
// set the owner in construct function
contract fixedOwner(){
    // owner is a constant.
    owner = 0xabcdef123456...;
    ...
}
```

Listing 7. Owner is a constant in the contract. Fuzzing engine cannot pass the *onlyOwner* modifier in this case.

Symbol n in the formula represents the current node and T_n is the set of all the target nodes. We define the distance between two nodes as the smallest number of hops from one node to the other. If node n is one of the target nodes, the distance is defined as 0. $d(n, t_n)$ represents the distance on the CFG between node n and the target node t_n. If node n is not one of the target nodes, the distance between a node and the target nodes is the harmonic mean of $d(n,t_n)$.

Let $t(s)$ be the trace of a seed. We can define the distance between a seed and the target nodes as:

$$D_s(s, T_n) = \frac{\sum_{n \in t(s)} D_n(n, T_n)}{|t(s)|} \tag{2}$$

Figure 6 shows an example of the distance between a node and the target nodes as well as the distance between a seed trace and the target nodes. The black nodes in the figure represents the target nodes. The number of each node shows the distance between the current node and the target nodes. The red path in the second graph represents the trace of a specific seed. For example, the distance between this trace to the target nodes is 1.34.

The seed that could decrease the distance to the target nodes will be reserved for the next round. In this way, the fuzzing engine can approach the target node rapidly. When the target is approached, the fuzzing process will stop, and the execution will be checked. If a protection mechanism is triggered, it is a false positive because the protection mechanism in the original contract would interrupt the execution. Otherwise, the function reported by the datalog analysis engine is unsafe. In other words, it is a true *Transfer In Tokens* problem.

The Algorithm 1 shows the working principle of the directed fuzzing engine. The algorithm's input contains the contract's source code and the target statements. The source code is generated by the compiler solc of the solidity smart contract. While the target statements are the suspect threats' locations given by the datalog engine. The pipeline of fuzzing is shown in function *FuzzAnalyze*. Line 2 to line 8 shows the preparation work. First, it generates a new piece of source code and a new CFG. Then Pied-Piper labels the target nodes in the CFG according to the location

ALGORITHM 1: Algorithms of the working pipeline of fuzzing engine in Pied-Piper

```
   Input: sCode: Source code of smart contracts
   Input: targets: target statements given by datalog analysis.
   Output: isThreat: Whether the function is a threat.
1  Function FuzzAnalyze(sCode,targets,duration):
2      newCode = _preprocess(sCode)
3      start = now
4      isThreat = true
5      newCFG = _constructCFG(newCOde)
6      LabelTarget(newCFG,targets)
7      calDistance(newCFG)
8      seeds.push(generateInitial(newCode))
9      while now - start < duration do
10         newseeds = []
11         for seed in seeds do
12             if ! (seedsDis = execute(seed, newCode)) then
13                 throw exception
14             if (seedsDis == 0) then
15                 isThreat = false
16                 return isThreat
17             if (chooseSeed(seed,seedsDis)) then
18                 newseeds.push(seed)
19         seeds = seeds.append(newseeds)
20         seeds = mutate(seeds)
21     return isThreat
22 End Function
```

reported in the datalog analysis and calculates the distance between each node to the target nodes. Finally, Pied-Piper generates an initial seed. Line 9 to line 21 describe the directed fuzzing process. For each seed in the seeds pool, Pied-Piper will execute the contract with it and collect the distance to the target nodes of each seed. As shown in lines 14 to 16, the fuzzing process will be stopped and eliminate the threat from the final report if the distance of the current seed to the target is 0. Pied-Piper will keep selecting the seeds by the distance and mutating the reserved seeds until the target is executed, as shown in lines 17 to 20. Specifically, the function 'chooseSeed' works like this: It first receives two parameters: 'seed' which represents the current seed and 'seedsDis' which means the distance between the current node trace and the target nodes. If the distance of current seed trace is shorter than before, it will be considered a good seed and be reserved in the pool for further mutation. The corresponding threat is reported as a true positive if the target is executed without triggering a protection mechanism.

6 IMPLEMENTATION AND EVALUATION

We implement Pied-Piper based on Vandal [4], which is a static program analysis framework for Ethereum smart contract and generates an intermediate representation for the contract. The fuzzing engine of Pied-Piper is implemented based on sFuzz [35]. In our evaluation, we seek to answer the following two research questions:

Table 3. Experimental Results on Manual Created Dataset

Arbitrary Transfer Problem	Datalog Analysis			with Dynamic Fuzzing		
	FP Samples	FN Samples	Avg Time	FP Samples	FN Samples	Avg Time
Exchange Tokens	0	0	5.61 s	0	0	\
Transfer In Tokens	7 (17.50%)	0	6.61 s	0	0	1 min 3.97 s
Transfer Out Tokens	0	0	6.08 s	0	0	\
Total	**7 (5.83%)**	0	6.10 s	**0**	0	1 min 3.97 s

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

RQ2. Is Pied-Piper efficient in detecting backdoor problems in real-world smart contracts?

6.1 Dataset and Environment Setup

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

- **Manually Created Dataset.** We prepared a dataset of 200 smart contracts[6] with certain types of backdoor problems. A backdoor function is manually embedded in each contract with the help of smart contract developers. Each type is embedded into 40 smart contracts.
- **Real-World Smart Contracts.** We wrote a crawler script to download the source code of smart contracts from Etherscan [13], a browser for Ethereum and smart contracts. In total, we got 13,484 real-world smart contracts to evaluate the effectiveness of Pied-Piper on real backdoor problem detection.

6.2 Accuracy on Backdoor Threats Detection

In order to check the performance of our datalog analyzer, we analyzed the characteristics and necessary components of a backdoor function and then constructed a small dataset consisting of 200 contracts that have been manually embedded with different types of backdoor functions. Initially, we only used the datalog analyzer to detect all the contracts and record the running time and number of false-positive and false-negative samples. Then we introduced dynamic testing to perform a second screening of suspicious functions that failed static analysis and recorded relevant information.

As shown in Table 3, we can find that the datalog analysis engine mislabelled seven samples as *Generate Tokens After ICO* type. Since the datalog analysis engine can only capture structural information and cannot make semantic level judgments, it will mark all functions that meet the component constraints as a backdoor, including some safety modes that meet the requirements of the ERC-20 standard. When equipped with a dynamic fuzzing engine, these functions will reach the terminal condition within a limited time, and the execution will be stopped. Then these kinds of misidentified samples will be corrected and removed from the suspicious set. **With the combination of datalog analyzing and directed fuzzing, Pied-Piper successfully reported all the 200 cases without any false-positive or false-negative errors.**

6.3 Efficiency on Real Smart Contracts

We evaluate Pied-Piper's efficiency in revealing backdoor threats in real-world smart contracts. On average, Pied-Piper uses 8.03 seconds (30.08 hours for all 13,484 contracts) to make static analysis

[6]The manually constructed contracts are listed at: https://github.com/EthereumContractBackdoor/PiedPiperBackdoor/tree/main/contracts/manualcontract.

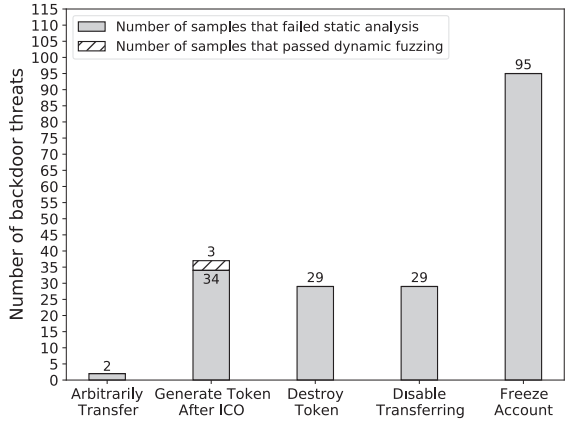

Fig. 7. Backdoor threats revealed by Pied-Piper. In total, there are 189 real threats found in all 13,484 contracts and three mislabeled samples are corrected by the dynamic fuzzer.

of each smart contract. The time required for the dynamic fuzzer is related to the artificially set threshold. In our experiment, the fuzzing duration is set to one minute. According to our previous experiments, functions in token contracts used for foundations during the ICO process usually stop within 40 seconds. So we use one minute as a threshold time limit for fuzzing. **In total, Pied-Piper reported 189 problems**[7] **and all of them are confirmed by the smart contract developers.** Among the 189 confirmed threats, four of them have been assigned 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 detailed results are shown in Figure 7.

The shadow bars in the figure represent the number of samples that are reported as a problem but eliminated by the fuzzing engine. From the result, we can see that *Freeze Account* is the most common type among these five types. The reason may be that many developers consider this type of function as a protection mechanism when an accident happens. If someone steals the tokens or cheats in a transaction, this function could be used as a reverting method to retrieve the loss. However, nobody can guarantee that this function will not be used in malicious situations. Besides, if the private key of the owner account is stolen [22], this kind of function may cause a disaster for all the users in this application. The *Arbitrarily Transfer* problem is not as common as the result shows. However, this kind of function may have the most severe impact (one of these kinds of function has caused a loss of $6.6 million as described in Section 3).

6.3.1 Real Threats Case Studies. In this section, we will give three real-world smart contracts as examples to illustrate how Pied-Piper detects the backdoor problems, which could hardly be found without automatic datalog analysis and the effectiveness of the dynamic fuzzing technique.

The first contract is deployed at address 0x0cb8d0b37c7487b11d57 f1f33defa2b1d3cfccfe. The source code is listed in Listing 8. This contract is special because there is neither *onlyOwner* modifier nor a transfer-like structure in it. However, it is judged as a contract with a *Transfer In Tokens* function 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 arbitrary transfer threat in the contract, which calls

[7]The contract with backdoor problems are listed at: https://github.com/EthereumContractBackdoor/PiedPiperBackdoor/blob/main/Backdoor_List.md.

another function in other contracts. This arbitrary transfer function could generate any amount of tokens to the owner's account, and anyone who has or steals the owner's private key could exploit this function to mint lots of tokens.

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

Listing 8. A contract with *Generate Token After ICO* problem. This contract calls a function which can mint tokens arbitrarily in another contract.

The second backdoor is found in contract *ComBillAdvancedToken* deployed at 0x6292CEc07c345C6c695 3e9166324f58db6D9F814. 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.

The function *approvedAccount* can freeze any account at any time. The function has an *onlyOwner* modifier which satisfies the rule *onlyOwner* of Figure 3. 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 3. Then, Pied-Piper could successfully trigger the rule *FreezeAccount* of Figure 4 and detect this backdoor successfully. There are 3,909 transactions made based on this contract before November 13, 2019, and this is a serious threat to all the accounts of this application.

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

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

The third case is found in contract which is deployed at address: 0x1966d718a565566e8e202792658d7b5ff4ece469 on Ethereum. The code is taken from the contract of nDEX token [34]. Listing 10 shows the related functions which are considered as a *Transfer In Tokens* problem by the datalog analysis engine. Function *adminClaimAirdrop* calls another function *doAirDrop*. This function has a *onlyOwner* modifier and the called function *doAirdrop* has a *Transfer In Tokens* structure. The static analysis engine can successfully detect this.

However, as line 6 shows, the distribution process can be stopped by the condition *total Distributed < totalSupply*. So this code describes a normal process of generating tokens in an ICO. Fuzzing engine can execute the function *doAirDrop* with the maximum value for parameter *_amount*. In the first execution, the variable *totalDistributed* will be set as a very significant value. In the next execution, the condition shown in line 6 will fail, and the execution will be interrupted. In this way, the fuzzing engine verifies that this is not a *Generate Token After ICO* problem and removes this case from the final report, and the false positive is eliminated.

```
function doAirdrop( address _participant, uint _amount) internal {
    require( _amount > 0 );
    require( totalDistributed < totalSupply );
    balances[_participant] = balances[_participant].add(_amount);
    totalDistributed =  totalDistributed.add(_amount);
    if (totalDistributed >= totalSupply) { distributionFinished = true;}
}

function adminClaimAirdrop(address _participant, uint _amount) public onlyOwner {
    doAirdrop(_participant, _amount);
}
```

Listing 10. Part of the contract code for nDEX token. Datalog analysis engine thinks there is a *Generate Token After ICO* problem here but fuzzing engine thinks there is not.

6.3.2 Time Overhead Analysis. Pied-Piper can analyze 13,484 contracts crawled from Etherscan in around 40.95 hours total, where 30.08 hours are spent on static analyzing and 10.87 hours on dynamic fuzzing. During the datalog analysis process, each contract takes an average of 8.03 seconds. For suspicious functions marked by the static analysis process, the dynamic fuzzer will pre-process the contract and deploy the contract in our local simulation environment, then generate seeds through target-oriented random mutation based on ABI information and call the suspected function. The time threshold of fuzzing is set to one minute. Suppose the input seed triggers the target statement within one minute and stops the ICO process. In that case, this function is determined to be a safe and necessary fundraising function instead of an arbitrary transfer problem.

In order to evaluate the efficiency of Pied-Piper's datalog analysis part, we also evaluate another datalog analysis tool for smart contracts: Securify [48]. As a result, Securify uses 115.18 hours to analyze all the contracts in our dataset. On average, Pied-Piper's datalog analysis is faster than Securify by about 73.89%. We did not compare Pied-Piper with Securify on the vulnerability detection's ability. This is because the datalog rules defined in Securify can only be used to detect bugs such as No Writes After Calls (Reentrancy). It has no rules such as 'transferwithoutSub' which are necessary for the backdoor detection. In fact, defining such rules to identify backdoor threats is one of the main contributions of Pied-Piper.

In summary, Pied-Piper can effectively detect real backdoor problems in Ethereum smart contacts. On the performance of embedded threats detection, both the false-positive and false-negative rates are zero. On the 13,484 real-world smart contracts, it detects 189 previous unknown potential threats. Furthermore, to avoid the impact of these problems, developers should standardize the development of smart contracts accordingly [17, 47] and control the group of accounts with too much power.

7 DISCUSSION & LIMITATIONS

Some threats to validity need to be discussed in our experimental evaluation.

Hard to Guarantee the Absolute Accuracy. Smart contract is a new technology, and there are no benchmarks with labelled smart contracts for the security research. Moreover, the problems we focus on have not been well studied. As a result, we can only manually check the experimental results' accuracy. However, manual checking is time-consuming and error-prone. In the evaluation part of the paper, we prepared two datasets for the experiment. We found that Pied-Piper has neither false positives nor false negatives on the manually created dataset. For the real-world smart contracts, we asked the smart contract developers to check the contracts reported with arbitrary

transfer problems by Pied-Piper. However, even so, we cannot guarantee that there are not any false negatives on real contracts.

Fairness of Manual Datasets. The first dataset used to evaluate the accuracy of Pied-Piper is embedded with arbitrary transfer problems manually. We built this dataset based on analysing representative contracts on Ethereum and the empirical study of existing threat reports. It may not contain all the possible situations of the threats. We consulted many developers of smart contracts to inject those threats, and this manual dataset is our best effort.

Feature or Bug of Backdoor Threats. The threats we discussed in this paper may have legitimate uses when an attacker is stealing some coins by some means. However, as we can see from the Soarcoin example, [40], this threat can be abused and cause a significant loss to regular users. Besides, it is hard to tell whether the owner or the hacker who stole the private key took advantage of these high authority functions. We think the developers of smart contracts should try their best to secure the code rather than develop high-risk remedial measures. Furthermore, to avoid these problems, developers can standardize the development of smart contracts accordingly [17, 47] and control the group of accounts.

8 RELATED WORK

Validation for Smart Contracts. Smart contracts have been increasingly used but also shown to contain many vulnerabilities [1, 6, 19]. Once these codes are deployed on Ethereum, the attacker can easily exploit them to launch an attack, causing significant 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 [33]. Due to data immutability, developers have limited ability 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, dynamic, and hybrid. Zeus [24] and FSolidM [32] 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 [25, 28, 36] apply symbolic execution to make up for this weakness, finding potential security bugs during execution. However, these tools only focus on the inner-contract vulnerabilities. To handle this problem, a recent work named Pluto [30] tries to construct inter-contract CFG to detect the hidden bugs in contact calls. Another dynamic method, like fuzzing, has been applied to verifying the contract, like Reguard [27], V-Gas [29], and ContractFuzzer [23]. Echidna [7] generates random call sequences based on a list of invariants for developers to check whether there is any problem. ILF [18] uses imitation learning to learn a fuzzing policy from a symbolic execution expert to guide the fuzzing process. SmartBugs [14] integrates several well-used tools to give a more detailed report on smart contract vulnerabilities. The authors of another tool SCStudio [38] also assembles commonly-used smart contract vulnerabilities detection tools to secure smart contracts with a relatively low false positive and false negative. They chose the basic analyzers based on an empirical study [39] which compares all these tools on their performance.

Declarative Program Analysis. Using a declarative language for program analysis will bring many 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 [20], for defining recursive relations. For example, the Doop framework [43] uses a logic-based language for the points-to analysis of Java programs and unequivocally redefines the state-of-art in pointer analysis, proving that Datalog is practical and effective in program analysis. Zhang et al. [52] present an effective method to find a relevant abstraction for program analyses written in Datalog, and authors in [31] also present Flix to specify and solve least

fixed point problems. As a variant of Datalog engines, Souffle [45] is designed for tool designers crafting static analyses and provides the ability to a rapid prototype.

For smart contracts, there are also some analysis tools based on Datalog. Vandal [4] is a static program analysis framework for smart contracts. It converts the EVM bytecode into an equivalent intermediate representation and feeds it to the Datalog engine. Similarly, MadMax [16] uses Datalog analysis to explore out-of-gas vulnerabilities within smart contracts. They define the identification rules for three types of out-of-gas vulnerabilities. Securify [48] also uses datalog analysis to test smart contract bugs. For now, Securify supports the detection of 38 types of vulnerabilities.

Main Differences. Unlike the above work, we try to guarantee the security of smart contracts from another perspective and develop an efficient method to detect security risks caused by super authority, especially for the backdoor threats. Our work uses the same inference engine as all Datalog-based work but leverages its advance for preliminary detection of arbitrary transfer threats. Therefore, when any improvement or enhancement for the Datalog-based program analysis technique is proposed, our tool will also benefit from it. Furthermore, the directed fuzzing technique is also customized to eliminate the false positives of datalog analysis. The proposed customization could also be applied to other datalog analysis-based work such as MadMax and Securify.

9 CONCLUSION

In this work, we propose Pied-Piper, a hybrid analysis tool to hunt the backdoor threats in Ethereum ERC token contracts. We first formulate the five common types of backdoor problems in smart contracts with a detailed empirical study. There are generally two ways to exploit these problems, which would lead to a significant loss of other users. The first one is that some malicious owners of the contract may use the high-privileged functions to meet their profits. The second one is by acquiring the private key of the contract owner. Both methods break the rule of decentralization and may destroy the user's trust in the whole system. Then, we designed and implemented Pied-Piper. Based on Datalog analysis and dynamic directed fuzzing, Pied-Piper could successfully hunt all the common types of backdoor problems. In the real-world smart contracts deployed on Ethereum, Pied-Piper successfully found 189 confirmed backdoor threats, with four of them having already been assigned with CVE identifiers in NVD. Our future work will focus on two aspects. First, we will try to polish our method to support more types of problems. Then, we will integrate the automatic problem repair to fix the detected threats.

REFERENCES

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

[2] Bancor. 2022. SmartToken. https://etherscan.io/address/0x1f573d6fb3f13d689ff844b4ce37794d79a7ff1c#code. Accessed April 4, 2022.

[3] Bitcoin. 2022. Bitcoin is an innovative payment network and a new kind of money. https://bitcoin.org/en/. Accessed June 16, 2022.

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

[5] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. {EnFuzz}: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1967–1983.

[6] ConsenSys. 2022. smart-contract-best-practices. https://github.com/ConsenSys/smart-contract-best-practices/blob/master/docs/known_attacks.md. Accessed April 4, 2022.

[7] crytic. 2022. echidna. https://github.com/crytic/echidna/.

[8] CVE. 2022. CVE-2018-1000203. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000203. Accessed April 4, 2022.

[9] CVE. 2022. CVE-2022-16944. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-16944. Accessed April 4, 2022.
[10] eric.eth. 2022. 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 April 4, 2022.
[11] Ethereum. 2022. Ethereum is the community-run technology powering the cryptocurrency ether (ETH) and thousands of decentralized applications. https://ethereum.org/en/. Accessed June 16, 2022.
[12] Ethereum. 2022. Ethereum/Solidity. https://github.com/ethereum/solidity. Accessed April 13, 2022.
[13] Etherscan. 2022. Etherscan. https://etherscan.io/. Accessed April 13, 2022.
[14] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A framework to analyze solidity smart contracts. *arXiv preprint arXiv:2007.04771* (2020).
[15] Hard Fork. 2022. PAX stablecoin has backdoor for freezing and seizing cryptocurrency. https://thenextweb.com/hardfork/2018/09/20/stablecoin-backdoor-law-enforcement/. Accessed April 4, 2022.
[16] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL* 2 (2018), 116:1–116:27.
[17] Gilad Haimov. 2022. How to Create an ERC20 Token the Simple Way. https://www.toptal.com/ethereum/create-erc20-token-tutorial. Accessed April 4, 2022.
[18] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2022. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 531–548.
[19] Yoichi Hirai. 2016. Formal verification of deed contract in Ethereum name service. November-2016. [Online]. Available: https://yoichihirai.com/deed.pdf (2016).
[20] Neil Immerman. 1999. Descriptive complexity. In *Graduate Texts in Computer Science*.
[21] Investopedia. 2022. 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 April 4, 2022.
[22] ISE. 2022. Ethercombing: Finding Secrets in Popular Places. https://www.ise.io/casestudies/ethercombing/. Accessed April 4, 2022.
[23] Bo Jiang, Ye Liu, and WK Chan. 2018. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.
[24] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing safety of smart contracts. In *NDSS*.
[25] Ao Li and Fan Long. 2018. Detecting standard violation errors in smart contracts. *ArXiv* abs/1812.07702 (2018).
[26] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. PAFL: Extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 809–814.
[27] Chao Liu, Han Liu, Zhao Cao, Zhuotong Chen, Bangdao Chen, and A. W. Roscoe. 2018. ReGuard: Finding reentrancy bugs in smart contracts. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)* (2018), 65–68.
[28] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. *IACR Cryptology ePrint Archive* 2016 (2016), 633.
[29] Fuchen Ma, Meng Ren, Fu Ying, Wanting Sun, Houbing Song, Heyuan Shi, Yu Jiang, and Huizhong Li. 2022. V-Gas: Generating high gas consumption inputs to avoid out-of-gas vulnerability. *ACM Transactions on Internet Technology (TOIT)* (2022).
[30] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2021. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering* (2021).
[31] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to Flix: A declarative language for fixed points on lattices. In *PLDI*.
[32] Anastasia Mavridou and Aron Laszka. 2017. Designing secure Ethereum smart contracts: A finite state machine based approach. In *Financial Cryptography*.
[33] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. 2019. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *J. Cases on Inf. Techn.* 21 (2019), 19–32.
[34] nDEX. 2022. nDEX token. https://etherscan.io/token/0x1966d718a565566e8e202792658d7b5ff4ece469. Accessed April 4, 2022.
[35] T. D. Nguyen, L. H. Pham, Jun Sun, Yun Lin, and Q. Minh. 2020. sFuzz: An efficient adaptive fuzzer for solidity smart contracts. *ArXiv* abs/2004.08563 (2020).

[36] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC*.
[37] OpenZeppelin. 2022. ERC 20. https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#IERC20-approve-address-uint256-. Accessed June 16, 2022.
[38] Meng Ren, Fuchen Ma, Zijing Yin, Ying Fu, Huizhong Li, Wanli Chang, and Yu Jiang. 2021. Making smart contract development more secure and easier. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1360–1370.
[39] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. 2021. Empirical evaluation of smart contract testing: What is the best choice?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 566–579.
[40] Mauro Sacramento. 2022. Backdoor Flaw Sees Australian Firm Lose $6.6 Million in Cryptocurrency. https://finance.yahoo.com/news/backdoor-flaw-sees-australian-firm-115323212.html. Accessed April 4, 2022.
[41] Allen Scott. 2022. New Research Finds Backdoor 'Centralized Control' in Many ICOS. https://bitcoinist.com/icos-centralized-control-new-study/. Accessed April 4, 2022.
[42] Sead. 2022. 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 April 12, 2022.
[43] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for fast and easy program analysis. In *Datalog*.
[44] SoarLab. 2022. SoarCoin. https://etherscan.io/address/0xD65960FAcb8E4a2dFcb2C2212cb2e44a02e2a57E#code. Accessed April 4, 2022.
[45] Souffle. 2022. Souffle. https://souffle-lang.github.io/index.html. Accessed April 4, 2022.
[46] SPAcoin. 2022. SpaCoin. https://etherscan.io/address/0x61402276c74c1def19818213dfab2fdd02361238. Accessed April 4, 2022.
[47] StackExchange. 2022. 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 April 4, 2022.
[48] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.
[49] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. 61–64.
[50] Udi Wertheimer. 2022. 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 April 4, 2022.
[51] CryptoGlobe Staff Writer. 2022. 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 April 4, 2022.
[52] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *PLDI*.

Received 13 April 2022; revised 18 June 2022; accepted 14 August 2022