TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum

Ting Chen
University of Electronic Science and Technology of China (UESTC)
China
brokendragon@uestc.edu.cn

Yufei Zhang
UESTC
China
2235285714@qq.com

Zihao Li
UESTC
China
gforiq@qq.com

Xiapu Luo*
The Hong Kong Polytechnic University
Hong Kong
daniel.xiapu.luo@polyu.edu.hk

Ting Wang
Pennsylvania State University
USA
inbox.ting@gmail.com

Rong Cao
UESTC
China

Xiuzhuo Xiao
UESTC
China

Xiaosong Zhang
UESTC
China

ABSTRACT

Motivated by the success of Bitcoin, lots of cryptocurrencies have been created, the majority of which were implemented as smart contracts running on Ethereum and called tokens. To regulate the interaction between these tokens and users as well as third-party tools (e.g., wallets, exchange markets, etc.), several standards have been proposed for the implementation of token contracts. Although existing tokens involve lots of money, little is known whether or not their behaviors are consistent with the standards. Inconsistent behaviors can lead to user confusion and financial loss, because users/third-party tools interact with token contracts by invoking standard interfaces and listening to standard events. In this work, we take the *first* step to investigate such inconsistent token behaviors with regard to ERC-20, the most popular token standard. We propose a novel approach to automatically detect such inconsistency by contrasting the behaviors derived from three different sources, including the manipulations of core data structures recording the token holders and their shares, the actions indicated by standard interfaces, and the behaviors suggested by standard events. We implement our approach in a new tool named `TokenScope` and use it to inspect all transactions sent to the deployed tokens. We detected 3,259,001 transactions that trigger inconsistent behaviors, and these behaviors resulted from 7,472 tokens. By manually examining all (2,353) open-source tokens having inconsistent behaviors, we found that the precision of `TokenScope` is above 99.9%. Moreover, we revealed 11 major reasons behind the inconsistency, e.g., flawed tokens, standard methods missing, lack of standard events, etc. In particular, we discovered 50 unreported flawed tokens.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; Use https://dl.acm.org/ccs.cfm to generate actual concepts section for your paper;

KEYWORDS

Ethereum, token, ERC-20, inconsistent behavior

ACM Reference Format:
Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. `TokenScope`: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19), November 11–15, 2019, London, United Kingdom.* ACM, 18 pages. https://doi.org/10.1145/3319535.3345664

1 INTRODUCTION

Motivated by the success of Bitcoin, lots of cryptocurrencies have been created. Since only a few cryptocurrencies are native assets (e.g., Bitcoin) of blockchains, the majority of them, so-called *token*s, are implemented as smart contracts running on Ethereum [16], because Ethereum is the largest blockchain that supports smart contracts. We will use the terms token and token contract interchangeably. These tokens usually involve lots of money. For example, the top 10 tokens on Ethereum are worth more than 2.8 billion USD [16]. To regulate the interactions between token contracts and users as well as the third-party tools (e.g., wallets, exchange markets, blockchain explorers), several standards have been proposed for the implementation of token contracts.

These standards usually specify *standard interfaces* (i.e., methods) as well as their functionalities, which should be implemented by token contracts, and *standard events* that should be emitted by token contracts to notify other applications. For example, ERC-20, the most popular token standard, defines 6 standard interfaces (we do

*The corresponding author.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3345664

not consider the optional standard interfaces) and two standard events [62].

Users usually employ third-party tools to manipulate tokens. For example, they use wallets to transfer tokens, leverage exchange markets to purchase/sell tokens, and employ blockchain explorers to check transactions. These tools interact with tokens through the standard interfaces and standard events defined in the token standards. For example, by investigating the source code of 10 popular third-party tools, we find that all of them recognize token behaviors by monitoring standard interfaces and/or standard events. More specifically, 3 blockchain explorers (i.e., EthVM [8], toy-block-explorer [7] and ETCExplorer [13]) and 1 wallet (i.e., MetaMask [37]) monitor standard methods. Ethereum ETL [22], a data collection tool, recognizes tokens by detecting standard interfaces, and captures token transfer behaviors by monitoring standard events. Moreover, 1 blockchain explorer (i.e., BlockScout [47]), 2 wallets (i.e, MyEtherWallet [38] and Etherwall [18]), and 2 exchange markets (i.e., EtherEx [15] and openANX [41]) monitor both standard methods and standard events.

However, if the implementation of token contracts is not consistent with the standards, third-party tools can neither interact with tokens properly nor even recognize tokens.

Unfortunately, little is known *whether the behaviors of the deployed tokens are consistent with the standards*. Inconsistent behaviors can lead to user confusion and financial loss. For instance, the token named blockwell.ai KYC Casper Token emitted standard events informing others that the tokens have been transferred. However, it did not really transfer the token and thus cheat users [64]. As another example, the token named USDT made fake deposits by invoking standard interfaces but did not transfer the tokens. The exchange markets mistakenly thought that some tokens were deposited because they detect token transfers by monitoring the invocation of standard interfaces [40].

In this work, we take the first step to investigate such inconsistent token behaviors with regard to ERC-20, the most popular token standard. Although some formal verification techniques have been proposed for checking the properties of smart contracts [2, 32, 49], it is very challenging for them to conduct such automated inspection, because they require developers to manually define the correct properties and specify all the code that is responsible for such properties in smart contracts. Unfortunately, since only less than 1% deployed smart contracts are open-source [19], it is difficult for analysts to locate all the code relevant to the defined properties. To the best of our knowledge, none of the existing studies on smart contracts examines the inconsistent token contracts [3, 20, 23, 36, 39, 57–60]. The closest work is from Fröwis et al. who propose two methods to recognize token contracts by analyzing Ethereum virtual machine (EVM) bytecode [20]. The first method relies on the method IDs of standard interfaces. However, they acknowledged that this method is prone to both false positives and false negatives [20]. For example, a false positive will be generated if a constant in the smart contract is equal to the ID of a standard interface [20]. Moreover, a false negative will be raised if a token implements standard methods in multiple contracts. The second method applies symbolic execution and taint analysis to detect the pattern of token transfers [20]. If a pattern is detected, a token contract is recognized [20]. More precisely, it applies taint analysis to check whether storage operations are determined by inputs, and uses symbolic execution to match path constraints with the symbolic expressions of the written values [20]. Unfortunately, this approach suffers from the limitations of symbolic execution and their pattern definition, which lead to false negatives [20]. Moreover, these two methods cannot detect token transfer behaviors realized by the cooperation of multiple contracts [20]. It is worth noting that their work aims to recognize tokens, but our work focuses on detecting inconsistent behaviors.

We propose a novel approach to automatically detect the inconsistent behaviors by contrasting the information from three different sources, including the manipulations of core data structures recording the token holders and their shares, the actions indicated by standard interfaces, and the behaviors suggested by the standard events. If any two of them do not match, an inconsistent behavior is detected. For example, an inconsistency happens if the token balance of a token holder is decreased by 10 whereas the standard `Transfer` event suggests a different amount. It is nontrivial to realize this approach because of two challenges: (1) how to automatically identify the core data structures that store each token holder's identifier and balance; (2) how to recognize token transfers that are triggered through inter-contract invocations. For example, when *user1* wants to transfer tokens to *user2*, the token contract can realize such functionality by calling one smart contract to decrease the balance of *user1* and then invoking another smart contract to increase the balance of *user2*. Such contract interaction hinders existing static analysis approaches from recognizing the token behaviors because it is difficult to know which contract will be invoked without runtime information.

To address these challenges, our trace-based approach leverages the salient feature of blockchain that the execution of all smart contracts can be restored from the blockchain. First, we recover the execution traces of token contracts by node instrumentation (§4.2) for investigating contract interactions. Second, we locate the core data structure that maintains each token holder's identifier and balance, and recognize the token behaviors in terms of manipulating the core data structure by exploiting how EVM accesses the data structures (§4.3). Third, we collect the token behaviors indicated by standard interfaces and the behaviors suggested by the standard events through parsing traces, and detect inconsistent behaviors by contrasting the information from the three sources (§4.4).

We implement our approach in a new tool named `TokenScope` and use it to inspect all transactions sent to all deployed tokens. `TokenScope` detects 3,259,001 transactions that trigger inconsistent behaviors, which are produced by 7,472 inconsistent tokens (§5). By manually examining all open-source (2,353) tokens exposing inconsistent behaviors, we find only 1 false positive, and thus the precision of `TokenScope` is above 99.9% (§5). Besides, we obtain several interesting observations from the experimental results. For example, 81% of inconsistent tokens were deployed after the finalization of ERC-20 standard, 1/3 of traded tokens are inconsistent tokens, and 17.6% of inconsistent tokens are traded in exchange markets. Moreover, we conduct a thorough investigation to reveal 11 major reasons behind inconsistent behaviors, including flawed tokens, standard methods missing, lack of standard events, etc (§6). In particular, we discover 50 unreported flawed tokens.

In summary, this work has three major contributions.

• To the best of our knowledge, it is the *first* work on detecting inconsistent token behaviors with regard to token standards. Our novel approach automatically detects the inconsistent behaviors by contrasting the information obtained from three different sources.
• We implement our approach in a new tool named `TokenScope` after tackling several challenging issues.
• Using `TokenScope` to inspect all transactions sent to all deployed token contracts, we found 3,259,001 inconsistent behaviors that resulted from 7,472 tokens and obtained many interesting observations. By manually examining all open-source tokens exposing inconsistent behaviors, we discover 11 major reasons for inconsistency and find that `TokenScope` has a very high precision.

2 BACKGROUND

Account. There are two types of accounts in Ethereum: external owned account (EOA) and smart contract. Only the smart contract accounts have executable code and they can be created by an EOA or another smart contract.

Smart contract. After being developed by any high-level languages (e.g., Solidity) and compiled into EVM bytecode, smart contracts will be deployed to the blockchain and executed by EVM according to the predefined program logic [63]. After deployment, a smart contract cannot be modified [63]. A smart contract can provide methods to be invoked by others, and emit events to inform other applications. When executing a smart contract, EVM maintains a runtime stack, the *memory* which is a transient space, and the *storage* which is a permanent space for storing data [63]. To prevent abusing resources, the deployment and invocation of a smart contract will charge money from transaction senders [63].

Transaction. A transaction is a message sent by an account. To invoke a smart contract, an account sends a transaction to the contract, which specifies the invoked method and carries parameters. There are two types of transactions depending on the senders of transactions, namely *external transactions* whose senders are EOAs, and *internal transactions* whose senders are smart contracts. Note that only the external transactions are stored in the blockchain. Although a smart contract can be called by another one, the first smart contract in this call chain should be invoked by an EOA.

Token A token is a smart contract which records the information of token holders and their shares, and supports token activities, e.g., query the balance of a token holder, transfer tokens to another holder. A token contract should implement standard interfaces and standard events so that other applications can interact with it. As the semantics of standard interfaces and standard events are well-specified in token standards, users know which standard method should be invoked to accomplish a task and can get the execution result by monitoring standard events.

Token contracts can also have non-standard interfaces and non-standard events, whose semantics are not specified in token standards. Like fiat money, a token has a total amount in circulation. *Token minting/burning* means increasing/decreasing the total amount, respectively.

ERC-20 standard. There are various token standards and the most popular standard is ERC-20 [62] which defines 6 standard method interfaces and 2 standard events. For example, the standard method `transferFrom`, declared as "function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)", transfers *_value* tokens from address *_from* to address *_to* [62]. Besides, `transferFrom`() must fire the standard event, `Transfer` [62]. This event is declared as "event Transfer(address _from, address _to, uint256 _value)", denoting that address *_from* transfers *_value* tokens to address *_to* [62]. Moreover, ERC-20 requires that the `Transfer` event should be emitted whenever tokens are transferred (no matter by standard methods or non-standard methods) [62]. This work focuses on 2 standard interfaces (i.e., `transfer`() and `transferFrom`()) and 1 standard event (i.e., `Transfer`) because they are related to the change of token balances.

Node & synchronization. The underlying structure of a blockchain is a P2P overlay that consists of multiple nodes. We only consider a *full* node because it implements all functionalities of Ethereum [63]. Each Ethereum node runs an EVM, and maintains the same copy of blockchain by synchronization. To reach the consensus with other nodes, besides downloading blocks from other nodes, each node replays all historical transactions to reach the same state. Hence, for each transaction sent to a contract, the contract will be executed in the node's EVM during its synchronization with other nodes.

3 MOTIVATING EXAMPLES

This section present two inconsistent tokens, one legitimate token, UGToken and one malicious token as motivating examples.

UGToken Fig. 1 shows the code of three functions in UGToken. All code in this paper are in Solidity, the most popular language for developing Ethereum smart contracts. UGToken uses a mapping variable, *balances*, to store the information of token holders, which maps the address of a token holder to the amount of the holder's tokens. Each function leads to an inconsistent behavior. Line 2 checks if the token holder *_from* has sufficient tokens to send. If not, the execution of the smart contract halts. However, Line 2 contains an integer overflow bug so that it can be bypassed if $\_feeUgt + \_value > 2^{255} - 1$, because both *_feeUgt* and *_value* are 256-bit unsigned integers. If integer overflow happens, the two token recipients (i.e., *_to* at Line 3, *msg.sender* at Line 5) will receive much more tokens than the amount sent by the token sender (i.e., *_from* at Line 7). Therefore, an inconsistency happens because the behavior indicated by the standard events (Lines 4, 6) does not reflect the real token behavior. Specifically, Lines 4 and 6 suggest that a huge number of tokens are sent by *_from*, however, *_from* just sends a few tokens due to integer overflow. The function `transfer`() will incur an inconsistent behavior after Line 10 executes, because `transfer`() is a standard interface suggesting that the account *msg.sender* will send *_value* tokens to *_to*, but no token will be transferred in practice. Such inconsistency is called fake deposit that could cheat exchange markets [40]. The function allocateToken() also leads to an inconsistency since Line 12 increases the balance of *owner*, but no `Transfer` event is emitted to announce such token minting behavior. Lacking of standard events can confuse third-party tools such as wallets, exchange markets, blockchain explorers, because they will not be informed when token transfers.

By replacing Lines 2, 3, 5, 7, 10, and 12 with 2*, 3*, 5*, 7*, 10*, and 12*, respectively, we can remove the inconsistency and the revised token contract complies with ERC-20. To fix the integer overflow, we import the SafeMath library which halts execution if an integer overflow happens [42]. To fix `transfer`(), we throw an exception to halt execution if no token is transferred. Note that the effect of

```
1  function transferProxy(address _from, address _to,
     uint256 _value, uint256 _feeUgt, ...)...{
2      if(balances[_from] < _feeUgt + _value) throw;
  //2* if(balances[_from] < SafeMath.safeAdd(_feeUgt, _value)) throw;
     ...
3      balances[_to] += _value;
  //3* balances[_to] = SafeMath.safeAdd(balances[_to], _value);
4      Transfer(_from, _to, _value);
5      balances[msg.sender] += _feeUgt;
  //5* balances[msg.sender] = SafeMath.safeAdd(balances[msg.sender], _feeUgt);
6      Transfer(_from, msg.sender, _feeUgt);
7      balances[_from] -= _value + _feeUgt;
  //7* balances[_from] = SafeMath.safeSub(balances[_from],
       SafeMath.safeAdd(_feeUgt, _value));
     ...}
-------------------------------------------------------------------------
8    function transfer(address _to, uint256 _value) returns(bool success) {
9      if(...)
       ...
10     else {return false;}}
  //10* else {throw;}}
-------------------------------------------------------------------------
11   function allocateTokens(...){
       ...
12     balances[owner] += value;}
  //12* balances[owner] = SafeMath.safeAdd(balances[owner], value);
       Transfer(0, owner, value);}
```

Figure 1: An inconsistent token, the UGToken

executing a smart contract will be canceled if the execution halts abnormally [63], and hence the revised functions transferProxy() and `transfer`() do not produce inconsistent behaviors. To fix allocateTokens(), we first use the SafeMath library to prevent potential integer overflow, and then emit a `Transfer` event to announce token minting (i.e., "Transfer(0, owner, value)" meaning that *owner* receives *value* tokens and nobody's balance decreases).

Due to page limit, we just demonstrate how `TokenScope` detects the inconsistent behavior incurred by the integer overflow in three steps. First, it locates the core data structure (i.e., *balances*) that stores the information of token holders. Then, it monitors the modification of *balances* (Lines 3, 5, and 7) to learn real token behaviors. That is, the balance of *_to* increases by *_value*, the balance of *msg.sender* increases by *_feeUgt*, and the balance of *_from* decreases by $\_value + \_feeUgt - 2^{256}$ due to integer overflow. Moreover, it monitors the standard event emissions (Lines 4 and 6) to obtain the token behaviors indicated by the event `Transfer`: the balance of *_to* increases by *_value*, the balance of *msg.sender* increases by *_feeUgt*, and the balance of *_from* decreases by $\_value + \_feeUgt$ which is larger than 2^{256}. By comparing the real token behaviors with the behaviors indicated by `Transfer`, we detect the inconsistency.

A Malicious Token. We craft a token whose implementation (Fig. 2) violates ERC-20, to illustrate how the token can steal tokens from token holders without being noticed. This token contract uses a mapping variable, *balances*, to store the information of token holders (Line 2). *balances* maps the address of a token holder to the amount of tokens possessed by the holder. Line 4 declares the standard event, `Transfer`. Lines 7 to 13 implement the standard interface, `transfer()`. Both the `transfer()` interface and the `Transfer` event (Line 13) indicate that the transaction sender *msg.sender* transfers *_value* tokens to *_to*. However, the contract steals *fee* tokens from *msg.sender* and sends them to a hacker (Lines 9, 11). It also defines another mapping variable, *victim*, to record the amount of tokens that have been stolen from token holders (Lines 3, 12). The standard method, `balanceOf()` (Line 14) is expected to return the amount of tokens possessed by the queried account. However, it deliberately returns a fake value that is the summation of the real value and the amount of stolen tokens (Line 15) to hide its activity of stealing tokens. Hence, users cannot notice it by invoking `balanceOf()`. Inconsistent behaviors happen when invoking `transfer()` and `balanceOf()`, because the standard interfaces and the standard event do not reflect the real token behaviors.

```
1  contract malToken {
2    mapping (address => uint256) public balances;
3    mapping (address => uint256) public victim;
4    event Transfer(address, address, uint256);
5    uint public fee = 1;
6    address hacker = 0xa49f0136194b7cb37a0ebc18fb840ce64c75091d;
7    function transfer(address _to, uint256 _value) returns(bool){
8      require (balances[msg.sender] >= _value + fee);
9      balances[msg.sender] -= _value + fee;
10     balances[_to] += _value;
11     balances[hacker] += fee;
12     victim[msg.sender] += fee;
13     Transfer(msg.sender, _to, _value);}
14   function balanceOf(address _owner) constant returns (uint) {
15     return balances[_owner] + victim[_owner];}}
```

Figure 2: An inconsistent token that steals tokens

This contract can mislead Ethereum wallets (e.g., MetaMask [37]) and explorers (e.g., Etherscan [14]). We find that MetaMask returns the fake value computed by Line 15 instead of the real token balance. Therefore, users cannot notice token stolen using MetaMask. By examining the source code of Metamask, we observe that it invokes the method `balanceOf()` to query token balance. We also notice that Etherscan, which discloses neither its source code nor the technical details, returns the fake value of token balance. Moreover, a user cannot detect token stolen by checking the transactions displayed in Etherscan, because Etherscan just shows the value that the user expects to send. We find that Etherscan learns token transfer activities by listening to the `Transfer` event and hence it can be misled.

Our approach detects such inconsistent behaviors in three steps, and thus our approach can help pinpoint such token stolen behavior. First, it locates the core data structure (i.e., *balances*, Line 2) for storing the information of token holders. Then, it monitors the manipulation of *balances* (Lines 8 to 11) to learn real token behaviors. More precisely, the balance of *msg.sender* decreases by *_value* + *fee*, and the balance of *_to* and *hacker* increase by *_value* and *fee*, respectively. After that, by monitoring method invocations and event emissions, our approach obtains the token behaviors indicated by the method `transfer()` and the event `Transfer`. More precisely, the balance of *msg.sender* decreases by *_value* + *fee*, and the balance of *_to* and *hacker* increase by *_value* and *fee*, respectively. After that, by monitoring method invocations and event emissions, our approach obtains the token behaviors indicated by the method `transfer()` and the event `Transfer`. That is, the balance of *msg.sender* should be decreased by *_value* and the balance of *_to* should be increased by *_value*. By comparing the real token behaviors with the behaviors indicated by `transfer()` and `Transfer`, we detect the inconsistency.

4 TOKENSCOPE

4.1 Overview

Inconsistency. We let `M` represent the core data structure in a token contract for recording the information of token holders. Since the

token balance of a token holder denotes her asset, we focus on *the token behaviors that change token balances*. Let B denote such token behaviors, which consists of a series of tuples <t_holder, Δ value>. Each tuple means that the balance of the token holder t_holder changes by $\Delta value$. We let B_m and B_e denote the token behaviors learned from the standard interfaces and the standard events, respectively, and let B_r denote the real token behaviors that modify M. B_m is $\emptyset$, if an external transaction invokes a non-standard method, because the semantics of a non-standard method is unknown. B_e is $\emptyset$, if the execution of a token contract does not emit a standard event. B_r is $\emptyset$, if the execution of a token contract does not modify M. If an external transaction invokes a standard method, an inconsistency happens when any two of B_m, B_e, and B_r do not match. On the other hand, if an external transaction invokes a non-standard method, we detect an inconsistency when $B_e \neq B_r$ without considering B_m due to its unknown semantics.

Inconsistent token. Once an inconsistent behavior is detected, we first locate the smart contracts recorded in the trace. Note that each trace records the execution of all smart contracts triggered by one external transaction (§4.2). For each of such contracts, if it invoked the standard methods or emitted the standard events or modified M or stored M, we regard it as an *inconsistent token*. Such definition includes both the case of individual inconsistent contracts and the case where several contracts interact to realize the token behaviors.

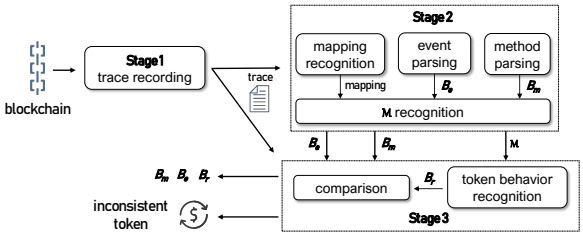

Figure 3: Architecture of TokenScope

Workflow. TokenScope consists of three stages (Fig. 3). It takes in the data from Ethereum, and outputs B_m, B_e, B_r for inconsistent token behaviors and the corresponding tokens. The first stage recovers the execution traces of all deployed smart contracts by instrumenting a full node. The second stage takes in the traces to recognize B_m, B_e by monitoring the invocations of standard methods and the emission of standard events, respectively, and then identifies M from the traces. This stage outputs M, B_m, and B_e as the input of the next stage. For each trace, the last stage recognizes B_r by monitoring the modification of M, and compares B_m, B_e and B_r to detect inconsistency. Note that the current implementation of TokenScope focuses on the change of token balances. That is, TokenScope detects the modification of M and monitors the emission of `Transfer` as well as the invocations of `transfer()` and `transferFrom()`. We will extend TokenScope to recognize other token behaviors (e.g., set the allowable amount of tokens that can be withdrawn by another account) in future work.

4.2 Stage 1: Trace Recording

A *trace* contains the execution log of smart contracts. An external transaction can trigger the execution of a smart contract, which may send several internal transactions to invoke other smart contracts. By recording the trace for each external transaction sent to a contract, we can get its execution log and that of internal transactions (as well as how a contract invokes others) if any. Each trace includes four parts, namely the hash of the corresponding external transaction, the address of the transaction receiver (i.e., the invoked smart contract), the data carried by the transaction which specifies the invoked method and parameters, and the executed EVM operations of all invoked smart contract in order. TokenScope instruments an Ethereum node to record traces since each node will download all blocks and replay all transactions during synchronization [63].

An approach to obtain traces is invoking the API, debug.traceTransaction() provided by an Ethereum full node, which takes in the hash of a transaction and outputs the trace triggered by that transaction [11]. However, there is no easy way to obtain all transaction hashes. Moreover, the API runs slowly because before executing the queried transaction it has to initialize the runtime environment, construct the correct state before the execution of the block containing the queried transaction, and then replay the preceding transactions before the queried transaction in the same block. Besides, APIs use Remote Procedure Calls to communicate with an Ethereum node, which introduces further delay. We compare the time required to collect traces between TokenScope with debug.traceTransaction(). The result is shown in Fig. 4. The x-axis means that we synchronize 200,000 blocks from the genesis block in the experiment. The cross in black color is the number of collected traces. Therefore, there are 8,674 traces collected from the first 200,000 blocks. The triangle in red color is the time consumption by invoking debug.traceTransaction(), and the point in blue color is the time consumption of TokenScope. We find that the difference between the time consumptions becomes larger when more blocks are downloaded. In particular, the API debug.traceTransaction() needs 8.7x time than TokenScope to collect the first 8,674 traces.

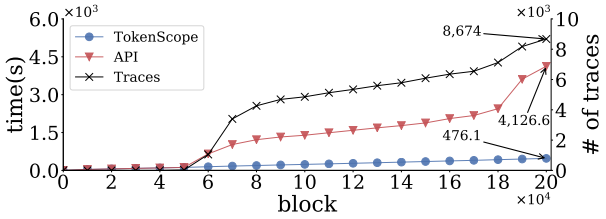

Figure 4: Time consumption to collect traces

Instead of using debug.traceTransaction(), TokenScope instruments an Ethereum node to record traces since each node will download all blocks and replay all transactions during synchronization [63]. To record the first three parts of a trace, we process external transactions and internal transactions in different ways. More precisely, to process an external transaction which starts the execution of a smart contract, we insert recording code into the function ApplyTransaction(), which is responsible for executing external transactions. Since each Ethereum node provides a handler for interpreting each EVM operation, to process an internal transaction that invokes a smart contract, we insert recording code into the handlers of CALL, CALLCODE, DELEGATECALL, and STATICCALL because these four operations generate internal transactions for invoking other smart contracts [63]. To record the last part (i.e., all executed EVM operations) of a trace, we insert logging code into the interpretation handler of each EVM operation to record the operation, values read by the operation, original values and

new values of the variables written by the operation. Taking the addition operation ADD [63] as an example, we record the operation, two addends, and the addition result.

To record contract interaction, we need to identify the address, the start and the end of each executed smart contract in the trace. We achieve this goal by maintaining a call stack. Specifically, when one of the handlers of CALL, CALLCODE, DELEGATECALL and STATICCALL is invoked, we obtain the address of the executed smart contract, which is the second item of the EVM stack, and then push the address on the call stack. When the handler returns, TokenScope pops the top item of the call stack. Therefore, we know the smart contract to which an executed EVM operation belongs by checking the top item of the call stack.

4.3 Stage 2: Locating Core Data Structure

Basic idea. The process of locating the core data structure for recording token information is presented in Algorithm 1. This stage takes in a trace, and outputs the core data structure M, token behaviors (i.e., B_m, B_e) suggested by standard methods and standard events, respectively. *Mapping* is a data structure that maps keys to values. It is a natural choice for storing the information of token holders, where the key is the identifier of a token holder and the value records the amount of tokens possessed by the token holder.

This stage consists of 4 steps. First, TokenScope locates all mapping variables in a contract, MAP (**Step 1**). Then, it recognizes the token behaviors learned from standard methods(B_m) (**Step 2**) and standard events (B_e) (**Step 3**) through trace analysis. Since a contract may use mapping variables to store other information, we exclude irrelevant mapping variables by correlating MAP with B_m and B_e. More precisely, we regard a mapping variable as M if two mapping items, whose keys are the two token holders specified by the standard methods or the standard events, are modified (**Step 4**), because standard methods and standard events reflect token transfer behaviors.

Algorithm 1: M recognition

Inputs: trace, t.
Output: Core data structure for maintaining token information, M;
Token behaviors suggested by standard methods, B_m;
Token behaviors suggested by standard events, B_e.

```
MAP = LocMap(t)                    //step1
Bm = ParseStandardMethods(t)       //step2
Be = ParseStandardEvents(t)        //step3
M = RecognizeM(MAP, Bm, Be)        //step4
return (M, Bm, Be)
```

Step 1: Locating Mapping variables. Without source code, locating mapping variables is challenging because there is no explicit mapping structure in EVM bytecode. To tackle the challenge, we exploit how a mapping variable is stored in EVM bytecode and how a mapping variable is manipulated by EVM operations. Note that all mapping variables are stored in the storage [63], and every variable stored in the storage has a unique 32-byte identity [23]. The EVM operations SLOAD and SSTORE are used to read and write data in the storage, respectively [63]. To access a mapping item which is a <key, value> pair, a SHA3 operation takes in the identity and the key to compute the location of the value. When executing a SHA3 operation, the identity is stored in a place specified by a stack item, and the key is stored before the identity.

We identify 4 types of mapping variables after manually inspecting all 16,248 open-source tokens that have been deployed on Ethereum and emitted the Transfer events. Although these 4 types might not cover all deployed tokens, how to automatically identify all types of mapping variables from the bytecode of smart contracts deserves another paper. We discuss a possible approach in §8 and will work on it in future work. In the following, we describe the 4 types of mapping variables and their accessing patterns.

Type-I. <key: *addr*, value: *amount*> This type of mapping associates the address of an account to the amount of her tokens. Fig. 5 shows how to read the amount from such a mapping variable with the source code (i.e., "amount = balances[addr];") and the corresponding EVM operations. The location of the value (*amount*) is the result of a SHA3 operation on the identity of *balances* and the key (*addr*). After that, the *amount* is read from the storage by a SLOAD.

Figure 5: Read *amount* from <key: *addr*, value: *amount*>

Type-II. <key: *addr*, value: *struct*> This type of mapping associates the address of an account to a *struct* that records the amount of tokens possessed by the account. Fig. 6 illustrates how to read the amount from such a mapping with the source code (i.e., "amount = balances[addr].amount;") as well as the corresponding EVM operations. The location of the value (*struct*) is the result of a SHA3 operation on the identity of *balances* and the key (*addr*). We discover that the *struct* is stored contiguously in the storage. Hence, an ADD operation is used to compute the location of *amount*, which is equal to the location of *struct* plus an *offset*. Finally, the *amount* is read by a SLOAD. If the *amount* is the first item of a *struct*, the ADD operation is not needed because the *offset* is 0 and the data structure becomes the same as Type-I shown in Fig. 5.

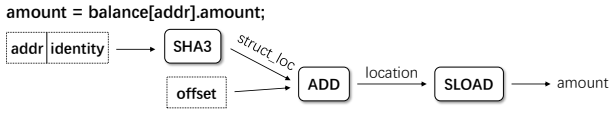

Figure 6: Read *amount* from <key: *addr*, value: *struct*>

Type-III. <key: *addr*, value: *struct*[]>. This type of mapping associates an array of *struct* to a token holder for recording all historical balances and the last item records her current token balance. We name such data structures as *checkpoints* and each item of *checkpoints* (i.e., a *struct*) as a *Checkpoint*. The array *checkpoints* is a global variable that has an identity and is stored in the storage [63]. By inspecting how EVM stores an array, we reveal that the identity refers to a storage location that stores the length of the array. Array items are stored contiguously, and the location of the first array item is the result of a SHA3 operation on the array's identity.

Fig. 7 presents the EVM operations of the source code "amount = balances[addr][balances[addr].length-1].amount;" for reading the current token balance. To ease the presentation, we use subscripts to differentiate multiple operations with the same opcode. The identity of the array *checkpoints* is the result of a $SHA3_1$ operation on the identity of *balances* and the address of a token holder. Since the array identity suggests the location that stores the array length, we get the length of *checkpoints* by a $SLOAD_1$. Since the array items are stored contiguously, the *item offset* of the latest *struct Checkpoint* from the first one is (length-1) × sizeof(Checkpoint). The size of

Checkpoint is a constant pre-computed during compilation of smart contracts. By adding the *item offset* to the location of the first *struct* which is the $SHA3_2$ result of the array identity, we get the location of the latest *struct*. By adding the *offset*, we pinpoint the location of the token balance, which is read by a $SLOAD_2$.

Our approach can also handle three special cases. First, to access the first *struct* of *checkpoints*, $SLOAD_1$, SUB and MUL are not needed because the *item offset* is 0. Second, when the balance of a token holder is modified, a new *struct* recording the latest token balance will be added to the array. In this case, the SUB is not needed because the *item offset* of the new *struct* is length × sizeof(Checkpoint). Third, if the *amount* is the first item of the *struct*, the ADD_2 is not needed because the *offset* is 0.

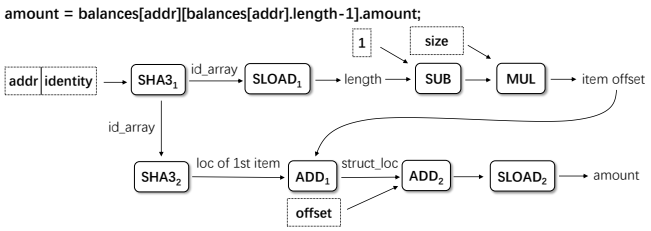

Figure 7: Read *amount* from <key: *addr*, value: *struct*[]>

Type-IV. Two maps. A smart contract can manage multiple kinds of tokens simultaneously, and one kind of token is associated with a *struct*, which is named as *Asset*. Such contract usually uses a mapping variable to associate the address of a token holder with an index, and we call this map variable as *holderIndex*. *Asset* contains a mapping variable that associates an index with a *struct* that records the token balance. We call the mapping variable as *wallets* and the *struct* as *Wallet*. Fig. 8 demonstrates how to access the balance with the source code "amount = Asset.wallets[holderIndex[addr]].amount;". The location of the index is the result of a $SHA3_1$ operation on the identity of *holderIndex* ($identity_1$) and the address of a token holder. Then, the index is read by a $SLOAD_1$. Similarly, the location of a *Wallet* is the result of a $SHA3_2$ operation on the identity of *wallets* ($identity_2$) and the index read from previous operations. The location of the amount is computed by adding an *offset* to the location of *Wallet*. Finally, the amount is read by a $SLOAD_2$. Our approach also handles a special case that the *amount* is the first item of the *struct*. In this case, the ADD is not needed because the *offset* is 0.

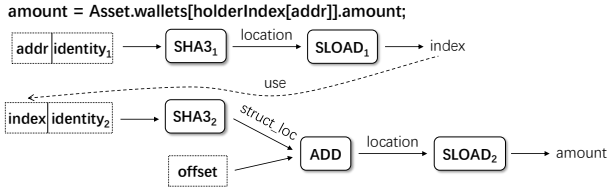

Figure 8: Read *amount* from *two maps*

Finding the identities of mappings. To locate a mapping variable, we need its identity. We locate it by conducting the def-use analysis [51] and leveraging the accessing pattern of mapping variables instead of searching the trace for SHA3 operations, because SHA3 can be used in other scenarios. We present the algorithm for finding mapping identities in more detail.

Algorithm 2 locates the 4 types of mapping variables from EVM bytecode, which takes in a trace and outputs the identities of mapping variables operated in the trace. We use the accessing pattern

Algorithm 2 : Mapping variables recognition

```
   Inputs: trace, t.
   Output: Identities of mapping variables and their types, ids.
1    for each op in t:
2      switch op:
3        case SHA3:
4          dep = isPara(t, op, res_manipulate)
5          if dep == true:
6            res_sha3_2 = getRes(t, op)
7            ids.remove(id);
8            id = getId(t, op) //the identity of wallets (Type-IV)
9            break
10         dep = isPara(t, op, res_sha3)
11         if dep == true: res_sha3_3 = getRes(t, op);
12         else:
13           reset();
14           res_sha3 = getRes(t, op);
15           id = getId(t, op) //the identity of the other 3 types
16       case SLOAD | SSTORE:
17         dep = isPara(t, op, res_add_2)
18         if dep == true: ids.append(id, 4); // Type-IV
19           dep = isPara(t, op, res_sha3_2)
20         if dep == true: ids.append(id, 4); // Type-IV
21         dep = isPara(t, op, res_add_4)
22         if dep == false: ids.changeType(id, 3); // Type-III
23         dep = isPara(t, op, res_add_3)
24         if dep == true: ids.changeType(id, 3); // Type-III
25         dep = isPara(t, op, res_add)
26         if dep == true: ids.append(id, 2); // Type-II
27         dep = isPara(t, op, res_sha3)
28         if dep == true: ids.append(id, 1); // Type-I, can be removed
29         res_manipulate = getRes(t, op)
30       case ADD:
31         dep = isPara(t, op, res_add_3)
32         if dep == true:
33           res_add_4 = getRes(t, op)
34           break
35         dep = isPara(t, op, res_sha3_3)
36         if dep == true:
37           dep = isPara(t, op, res_mul)
38           if dep == true: res_add_3 = getRes(t, op) //not first array item
39           else: res_add_4 = getRes(t, op) //first array item
40           break
41         dep = isPara(t, op, res_sha3_2)
42         if dep == true:
43           res_add_2 = getRes(t, op)
44             break
45         dep = isPara(t, op, res_sha3)
46         if dep == true: res_add = getRes(t, op)
47       case SUB:
48         dep = isPara(t, op, res_manipulate)
49         if dep == true: res_sub = getRes(t, op);
50       case MUL:
51         dep = isPara(t, op, res_sub)
52         if dep == true: res_mul = getRes(t, op) //do not add one item
53         else:
54           dep = isPara(t, op, res_manipulate)
55           if dep == true: res_mul = getRes(t, op); //add one array item
56   return ids
```

getRest(t, op): get the result of *op*.
isPara(t, op, res): whether a parameter of *op* is *res*. If *res* is null, this function returns false.
getId(t, op): get identity from a SHA3 operation.
reset(): delete all temporary variables (e.g., dep).

of Type-III shown in Fig. 7, which involves the most number of EVM operations compared to the other patterns, to explain the algorithm. For each EVM operation *op* in the trace *t*, if the operation is SHA3, Line 4 executes. The function isPara(t, op, res) applies def-use analysis to check whether *res* is a parameter of *op* in the trace *t*. If *res* is null, isPara() returns false. Hence, we skip Lines 5 – 11, since *res_manipulate* and *res_sha3* are null. The function reset() deletes all temporary variables (e.g., *dep*). If a mapping variable is found, we start to detect another mapping variable after executing reset() (Line 13). The result of SHA3, denoted by *res_sha3*, is obtained by the function getRes() (Line 14). A possible identity of a mapping variable, denoted by *id*, is obtained by the function getId() (Line 15). Whether *id* is a real identity will be checked in the following steps. Note that getRes() and getId() just parse the trace, because the parameters and results of all execution EVM operations have already been recorded in the trace.

When a SLOAD is executed (Line 16), the algorithm conducts several checks. We skip Lines 17 – 26 since *res_add_2*, *res_sha3_2*, *res_add_4*, *res_add_3*, and *res_add* are null. Since *res_sha3* is a parameter of the SLOAD, the check at Line 27 returns true. Then, *id* and the Type-I is added into the list *ids* (Line 28), indicating that we have found a mapping variable of Type-I. The result of SLOAD, *res_manipulate* is obtained at Line 29. When a SUB is executed (Line 47), our algorithm checks whether one of its parameters is *res_manipulate* (Line 48). If so, we get the result of the SUB, *res_sub* at Line 49. When a MUL is executed (Line 50), our algorithm checks whether *res_sub* is one of its parameters (Line 51). If so, the result of the MUL, *res_mul*, is obtained at Line 52. *res_mul* is actually the offset from the first item of the array.

When a SHA3 is executed (Line 3), the algorithm checks whether one of its parameters is the result of another SHA3 (Line 10). If so, the result of the SHA3, *res_sha3_3* (Line 11) should be the location of the first item of the array. When an ADD is executed (Line 30), several checks are conducted. We skip Lines 31 – 34, since *res_add_3* is null. Then, the algorithm checks whether the two parameters of the ADD are *res_sha3_3* and *res_mul* (Lines 35 – 37). If so, the result of the ADD, *res_add_3* is the location of the array item that should be read (Line 38). Then, an ADD is executed again. Our algorithm checks whether *res_add_3* is one of its parameters (Line 31). If so, the result of the ADD, *res_add_4*, is the location of token balance (Line 33). Finally, a SLOAD is executed to read the token balance. The algorithm checks whether *res_add_4* is one of the parameters of the SLOAD (Line 21). If so, the accessing pattern of Type-III is found, and we change the type of *id* from 1 to 3 (Line 22). The reason for changing the type is that the pattern of Type-I is a sub-pattern of Type-III. Hence, the algorithm first identifies the pattern of Type-I inside the pattern of Type-III.

Step 2: Parsing standard methods. We analyze `transfer()` and `transferFrom()`, which are used to transfer tokens, according to their semantics defined in ERC-20 [62]. Other token standards are discussed in §8. We detail how to monitor the invocation of `transfer()` and omit the processing of `transferFrom()` as it is similar. `transfer()` is defined as "function transfer(address _to, uint _value) public returns (bool success);", which allows the transaction sender to transfer *_value* tokens to the token holder *_to* [62]. Therefore, B_m has two tuples, <sender, -*_value*> and <*_to*, *_value*>. Since both external and internal transactions can invoke smart contracts, we handle them separately.

To handle external transactions, we insert recording code to the function TransitionDb(), which calls vmenv.Call() to execute a smart contract. Then, we obtain the transaction sender from the first parameter of vmenv.Call(), and get *_to* and *_value* that are placed together in the third parameter. To handle internal transactions, we instrument the interpretation handlers of CALL, CALLCODE, DELEGATECALL and STATICCALL. We only describe the instrumentation of opCall(), the interpretation handler for CALL, because other handlers are instrumented in a similar way. Since opCall() invokes env.Call() to execute a smart contract, we obtain the transaction sender from the first parameter of env.Call(), and acquire *_to* and *_value* from the third parameter.

Step 3: Parsing standard events. TokenScope interprets the `Transfer` event according to its semantics defined in ERC-20. First, we look for all logging operations (i.e., LOG0, LOG1, LOG2, LOG3, LOG4) from the trace since logging operations are responsible for emitting events [63]. For each logging operation, we get the third 32-byte value read by the operation because it is the event ID. Since each event has a unique ID that is the hash of the event signature [63], we locate the `Transfer` event according to its ID. After recognizing a `Transfer` event, we record B_e which includes two tuples <*_from*, -*_value*> and <*_to*, *_value*>. *_from*, *_to* and *_value* are the 4th – 6th values read by the logging operation, respectively.

Step 4: Recognizing the core data structure `M`. Since a token contract may have multiple mapping variables, we need to distinguish `M`, which stores the information of token holders, from irrelevant mapping variables. Our idea is to correlate the modification of a mapping variable with the standard interfaces and the `Transfer` event. More precisely, if there exists a trace where the modification of a mapping variable is accordant with the standard interfaces or the `Transfer` event, we regard the mapping variable as `M`. We detail how to correlate with the `Transfer` event as follows, and omit the correlation with the standard interfaces because they have a similar process. Since the `Transfer` event records two addresses (i.e., the sender and the receiver of tokens), we look for a mapping variable whose two items corresponding to the two addresses are modified. If found, the mapping variable is `M`. We use the token contract in Fig. 2 to illustrate how `TokenScope` distinguishes `M` (*balances*) from the irrelevant mapping variable (*victim*). From the `Transfer` event (Line 13), we get two addresses, *msg.sender* and *_to*. There are two mapping variables, *balances* (Line 2) and *victim* (Line 3). For *balances*, the two mapping values whose keys are *msg.sender* and *_to*, respectively, are updated (Lines 9, 10) when the method `transfer()` is executed. Therefore, we regard *balances* as `M`. In contrast, for *victim*, since only one mapping value corresponding to the key *msg.sender* is updated (Line 12), *victim* is not `M`.

Algorithm 3: Inconsistency Detection

Inputs: trace, t;
Core datastructure for maintaining token information, M;
Token behaviors suggested by standard methods, B_m;
Token behaviors suggested by standard events, B_e.
Output: Whether an inconsistency happens, bin.

```
Br = TokenBehavior(t, M)                          //step1
if Bm == null: bin = Match(Be, Br)                //step2
else bin = Match(Bm, Be, Br)                      //step2
return bin
```

4.4 Stage 3: Detecting Inconsistent Behaviors

As shown in Algorithm 3, taking in a trace and the outputs of stage 2, this stage detects inconsistency through two steps, namely recognizing real token behaviors B_r by monitoring the modification of `M` (**Step1**) and comparing B_m, B_e, and B_r (**Step 2**).

Step1: Token behavior recognition. B_r is a set of tuples, <*t_holder*, Δ value > for every trace. This step is similar to the first step in stage 2, because we locate mapping variables by exploiting their accessing patterns. For the ease of presentation, we describe this step by using the example in Fig. 2. After the execution of `transfer()`, `TokenScope` records three tuples, <*msg.sender*, -(*_value*+*fee*)>, <*to*, *_value*>, and <*hacker*, *fee*>. To detect the balance change of a token holder, we first look for a SHA3 operation, which reads the identity of `M` and the address of a token holder (i.e., the key), from each trace. Then, we check if the result of the SHA3 is read by a SSTORE through

the def-use analysis [51]. If so, we get the tuple $< bal^{old}, bal^{new} >$, where bal^{old} and bal^{new} are the original value and the new value written by the SSTORE, respectively. Since the balance of an account can be modified several times in a trace, we may find several such SHA3, and thus get n tuples, $< bal_i^{old}, bal_i^{new} >, 1 \leq i \leq n$. $\Delta value = bal_n^{new} - bal_1^{old}$, because the trace records the modifications of an account balance in order.

Step2: Comparison. We consider two cases. First, if $B_m \neq \emptyset$ indicating that an external transaction invokes a standard method, we compare B_m, B_e, and B_r. We find an inconsistency if any two of them do not match. Second, if $B_m = \emptyset$ indicating that an external transaction invokes a non-standard method, we only compare B_e and B_r because the semantics of the non-standard method are unknown, and we find an inconsistency if $B_e \neq B_r$. Reconsider the example in Fig. 2, assuming that an external transaction invokes the standard method `transfer()`, B_m and B_e are the same, which are <*msg.sender*, -*_value*> and <*_to*, *_value*>. However, B_r is different from them, which are <*msg.sender*, -(*_value*+*fee*)>, <*_to*, *_value*>, and <*hacker*, *fee*>. Therefore, TokenScope detects the inconsistent behavior and the token is considered as an inconsistent token.

Special addresses handling. The `Transfer` event uses special addresses to indicate special token behaviors. There are three special addresses, including 0, the address of the token contract, and the address of the account who creates the token contract (i.e., token creator). Consider a `Transfer` event "event Transfer(address _from, address _to, uint256 _value)", a token contract often sets *_from* to one of the special addresses to indicate token minting [12]. Similarly, a token contract often sets *_to* to one of the special addresses to indicate token burning [12]. In both cases, M will not be modified if the balances possessed by the special addresses are not recorded in M. To avoid false positives in detecting inconsistent behaviors, after finding a mismatch between B_e and B_r, we check whether it is because B_r does not contain the balance change of special addresses. If so, we do not consider such mismatch as inconsistency.

5 EXPERIMENTS

5.1 Results

To evaluate TokenScope, we download all 6,066,793 blocks from the launching of Ethereum (Jul. 30, 2015) to Aug. 1, 2018. We obtain all 7,123,729 deployed smart contracts as well as all 282,342,715 external transactions, and record 119,245,201 traces for all transactions sent to smart contracts.

Table 1 lists the numbers (before '/') of tokens, inconsistent tokens adopting different M types, and transactions triggering inconsistent behaviors. The figures after '/' denote the numbers of open-source tokens, open-source inconsistent tokens and the numbers of transactions triggering inconsistent behaviors of open-source inconsistent tokens. TokenScope detects 57,411 tokens, where 7,472 (13%) tokens are inconsistent and their inconsistent behaviors are triggered by 3,259,001 transactions. We find that 2,353 inconsistent tokens open their source code in Etherscan. Most tokens adopt M of Type-I, and 500+ tokens choose the other types. We find that 3,334 tokens present inconsistency when executing standard methods while 4,700 tokens show inconsistency when executing non-standard methods. Moreover, 562 tokens present inconsistency in both standard methods and non-standard methods.

Table 1: Tokens with different types of M

Type	# of tokens	# of inconsistent tokens	# of transactions
I	56,864/16,248	7,329/2,329	3,199,583/2,069,581
II	58/30	21/16	13,085/12,550
III	227/92	60/3	38,712/872
IV	262/92	62/5	7,621/465
sum	57,411/16,462	7,472/2,353	3,259,001/2,083,468

Deployment time. Fig. 9 shows the deployment time of inconsistent tokens, where each cross (x, y) indicates that there are y inconsistent tokens deployed in the period of x weeks after the deployment of the first inconsistent token. We find that the first inconsistent token was deployed on Nov. 26, 2015, nearly 3 months after the debut of Ethereum. ERC-20 was proposed on Nov. 19, 2015 and formally adopted on Sep. 11, 2017 after several revisions [54]. We observe that 81% $((7,472-1,420)/7,472)$ of inconsistent tokens were deployed after the finalization of ERC-20. Moreover, the number of inconsistent tokens increases steadily over time.

Remark1: there are still many inconsistent tokens after finalizing ERC-20. The gap between the description of ERC-20 and the understanding of token developers may be one root cause.

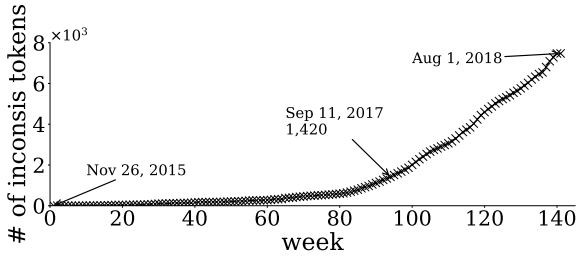

Figure 9: Deployment time of inconsistent tokens

Exchange markets. Table 2 lists the numbers of tokens traded in 5 centralized exchange markets and 4 decentralized exchange markets. The 2nd row shows market names and the 3rd row displays the number of tokens traded in each market. The 4th row aggregates the results of centralized markets and decentralized markets, separately, and the last row lists the aggregated result from these 9 markets. The numbers before and after '/' denote the number of tokens and the number of inconsistent tokens, respectively. We obtain the number of tokens traded in each centralized market by visiting its website because centralized markets usually maintain a list of traded tokens. We then get the number of inconsistent tokens traded in each centralized market by searching its website for the names and addresses of inconsistent tokens.

We adopt a different way to get the numbers for decentralized exchange markets (DEXs) because DEXs do not maintain the list of traded tokens. More precisely, we crawl the webpages of the decentralized exchange order tracker [17] that presents all transactions of DEXs, and then parse the collected transactions to get the addresses of all tokens traded in each decentralized exchange market. After that, we get the number of inconsistent tokens traded in each decentralized exchange market by matching the addresses of all traded tokens with the addresses of all inconsistent tokens. Results show that 1/3 $(1,314/3,947)$ of traded tokens are inconsistent and 17.6% $(1,314/7,472)$ of inconsistent tokens are traded in exchange markets. Note that all 348 tokens traded in centralized markets are also traded in decentralized markets.

Remark2: inconsistent tokens can incur severe financial consequences because many inconsistent tokens are traded in markets.

Table 2: Number of tokens traded in exchange markets

	Centralized exchange market					Decentralized exchange market			
	Binance	Bitfinex	Poloniex	Kucoin	Huobi	IDEX	EtherDelta	Token Store	Kyber Network
	177/19	99/16	19/10	172/9	25/3	1,349/499	3,848/1,248	219/91	52/20
∪	348/24					3,947/1,314			
∪	3,947/1,314								

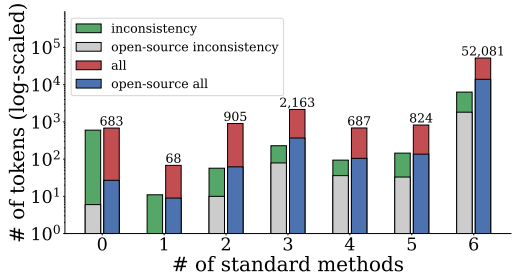

Figure 10: Number of standard methods realized in tokens

5.2 Token Transfers via Multiple Contracts

We classify tokens, open-source tokens, inconsistent tokens and open-source inconsistent tokens according to the number of standard methods implemented by them, and present the results in Fig. 10. ERC-20 requires to implement all six standard interfaces [62], but this figure shows that about 9.3% ($5,330 = 683 + 68 + 905 + 2,163 + 687 + 824$) tokens implement fewer standard methods.

It is interesting that 1.2% (683) tokens implement 0 standard methods. By investigating their source code (if any), bytecode and traces, we find that all of them transfer tokens via multiple contracts. For example, some developers deploy a token contract that implements standard token interfaces as a library (termed by *lib*). Then, other developers write a token contract (termed by *tc*) with customized functionality that loads *lib* and invokes the standard interfaces implemented by *lib* to transfer tokens. TokenScope can recognize such tokens that transfer tokens through multiple contracts, because it handles contract interaction by trace analysis. Moreover, 88% (601) of the tokens with 0 standard methods are inconsistent tokens. A possible reason is that it is more difficult to develop the tokens consisting of multiple contracts. Fig. 11 lists two inconsistent tokens detected by TokenScope where the contract (0x38cD) loads another contract (0x2a21) as a library. 0x2a21 implements all 6 standard interfaces while 0x38cD stores M and executes the code in 0x2a21 to manipulate M. TokenScope detects an inconsistency since 0x2a21 does not emit the `Transfer` event after transferring tokens. TokenScope considers both contracts as inconsistent tokens because the former stores M while the latter modifies M.

Besides loading library, token transfer can also be completed by inter-contract invocations. As an example, Fig. 12 presents 4 inconsistent tokens (in gray boxes) from a trace recording the execution of 5 smart contracts. In particular, an EOA *A* invokes the method withdrawToken() of the contract Etherdelta_2, then withdrawToken() invokes the method `transfer()` of the contract FunFair_Old. This method emits the `Transfer` event. Besides, this method invokes the method `transfer()` of the contract Controller, in which

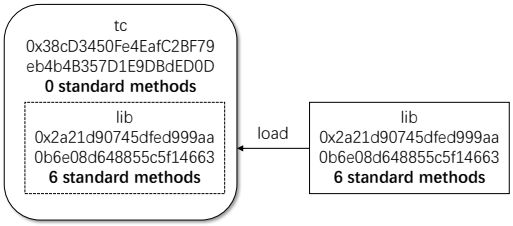

Figure 11: Two interacted inconsistent tokens

the method `transfer()` of the contract ledger is called. The latter `transfer()` manipulates M. After the execution of the contract ledger, the Controller calls the method controllerTransfer() of the contract FunFair, which also emits the `Transfer` event. A recent online discussion disclosed that FunFair is a new version of FunFair_Old, and each version emits the `Transfer` event when tokens are withdrawn from Etherdelta_2 [21]. Our approach discovers the inconsistency because the `Transfer` event is emitted twice.

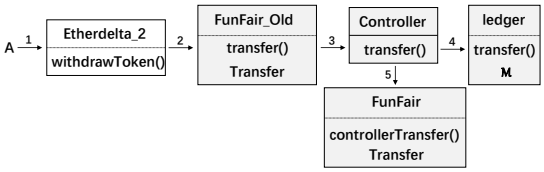

Figure 12: An inconsistency found by cross-contract analysis

5.3 Precision of TokenScope

We define precision as the ratio of the number of real inconsistent tokens to the number of inconsistent tokens discovered by TokenScope. A false positive refers to a token contract that complies with ERC-20 but is regarded as an inconsistent one by TokenScope mistakenly due to incorrect computation of either B_m, B_e, or B_r.

To evaluate the precision of TokenScope, we manually check all 2,353 open-source inconsistent tokens detected by TokenScope, and find only 1 false positive (i.e., MiniApps). Hence, the precision is $99.9\% = (2,353-1)/2,353$. Manual inspection reveals that MiniApps hardcodes an address in the contract and uses the address as the key to access M. Note that the location of the mapping value, whose key is the hardcoded address, will be pre-computed during compilation if full optimization is used. Consequently, SHA3 is not needed in the contract's bytecode to compute the location. TokenScope can locate M because MiniApps also accesses the balances of other token holders (not the hardcoded one). But it mistakenly reports an inconsistency when the balance of the hardcoded address is modified, because $B_r = \emptyset$ due to the lack of SHA3.

Screening through whitepapers. If the whitepaper of an inconsistent token describes the inconsistency, such inconsistency may not cause severe consequences because users can get aware of the inconsistency by reading the whitepaper. We further evaluate how many inconsistent tokens detected by TokenScope can be filtered out by their whitepapers. Without considering the false positive mentioned above, we search for the whitepapers of all 2,352 open-source inconsistent tokens from the Internet including the official websites of tokens, whitepaper collection websites and forums, and successfully download 752 whitepapers. After reading them, we find that only 31 whitepapers describe token behaviors in detail (e.g., how many tokens will be charged as fee) and only 1 token's

(i.e., Krown) whitepaper mentions the inconsistency: "An Event 'Authority Notified' is generated on the blockchain to notify Central Authority" [46] instead of the Transfer event. Note that TokenScope considers Krown as an inconsistent token because $B_e \neq B_r$. This analysis shows that existing whitepapers pay little attention to describing inconsistency.

Screening through other standards. TokenScope may detect an inconsistency if a token contract emits a standard event defined by other token standards, because that event is a non-standard event whose semantics is not defined in ERC-20. Such inconsistency may not introduce serious problems because users can get aware of the inconsistency by reading the other token standards. By checking all 2,353 open-source inconsistent tokens, we find that TokenScope detects 681 (0.017% = 681/3, 259, 001) inconsistent behaviors resulted from 10 (0.4%=10/2,353) tokens. These 10 tokens emit both the ERC-20 Transfer event and a new type of Transfer event defined by the ERC-223 standard, whose prototype is "event Transfer(address _from, address _to, uint256 _value, bytes _data)" [10]. Note that the ERC-223 Transfer event is different from the ERC-20 Transfer because they have different event prototypes. TokenScope identifies these 10 tokens since they emit ERC-20 Transfer events, and it detects inconsistent behaviors when these tokens emit ERC-223 Transfer events. These 10 tokens are listed in Table 3. We plan to extend TokenScope to support other token standards in future.

Table 3: Ten Inconsistent Tokens due to the Event Defined in ERC-223

Token name	Token address
Bodhi Ethereum	0x47c171cE16c1C06AaE6E785Ba3c518C42235da0F
WubCoin	0x2664877980f2684c9e9be07a50330e85847c5241
Tablow Club	0xeab447c1e2b5a76b57f15e55eab504801aa6ceb0
EIB	0x314d759476c5a3a02c0c6b1f1e213949084e277b
AnythingApp Token	0x36f74e50de0b79f9b0bbeb644af9d40e3cc26433
eDogeCoin	0x44cba3a62a15ac8f66ff75bf7abd058dcca7d7ed
Coinvest COIN Token	0x4306ce4a5d8b21ee158cb8396a4f6866f14d6ac8
Ethereum Lendo Token	0x45d0bdfDFBfD62E14b64b0Ea67dC6eaC75f95D4d
Test Token	0xfa74f89a6d4a918167c51132614bbbe193ee8c22
Auctus Token	0xfd89de68b246eb3e21b06e9b65450ac28d222488

5.4 Vetting Tokens before Deployment

Although TokenScope focuses on detecting inconsistent behaviors that have been happened in Ethereum, it can be easily extended to discover inconsistent tokens before deployment by equipping it with any path exploration techniques (e.g., symbolic execution, fuzzing) to generate traces. To demonstrate the feasibility, we develop a tool named TokenFuzzer that integrates TokenScope with ContractFuzzer, which is an open-source fuzzing tool for discovering security vulnerabilities of smart contracts [30], to detect inconsistent tokens. ContractFuzzer instruments the EVM to check whether security vulnerabilities are triggered, and runs the target smart contracts with generated inputs (i.e., transactions) in a private chain equipped with the customized EVM. TokenFuzzer reuses the code for generating transactions from ContractFuzzer, replaces the EVM instrumented by ContractFuzzer with the EVM instrumented by TokenScope for recording traces, and reuses the code for locating M and detecting inconsistent behaviors from TokenScope.

To evaluate TokenFuzzer, we first manually identify 20 inconsistent tokens from all open-source tokens that have not exposed inconsistent behaviors yet. That is, TokenScope has not discovered their inconsistent behaviors. It is worth mentioning that we do not select many inconsistent tokens to test TokenFuzzer due to two reasons. First, ContractFuzzer needs to run a private chain which is very time consuming. Second, the purpose of this experiment is to demonstrate that TokenScope can be extended to vet token contracts before their deployment, and we will further enhance TokenFuzzer's capability and boost its performance in future work.

The experimental results show that TokenFuzzer discovers 7 (35% = 7/20) inconsistent tokens, where 2 tokens have integer overflow bugs and the other 5 tokens do not emit standard events when M is modified (reasons are detailed in §6). After manually inspecting the 13 undiscovered inconsistent tokens, we find 3 issues: (1) some code can only be triggered by certain accounts; (2) some code can only be triggered when another method has already executed; (3) ContractFuzzer randomly generates inputs which are difficult to trigger integer overflow. To evaluate whether TokenFuzzer can discover such inconsistent tokens if these 3 issues are solved, we use proper accounts to invoke token contracts, properly arrange the order of the tested methods, and modify the input generation strategy of ContractFuzzer to produce desired values. After that, we re-run TokenFuzzer and find that it can successfully discover all these inconsistent tokens.

Remark3: TokenScope can be easily extended to discover inconsistent tokens before token deployment (i.e., inconsistent behaviors have not been trigger yet) if it is equipped with a proper path exploration component. We will investigate it in future work.

Table 4: 11 major reasons for inconsistency

Reason	#	Description
Flawed tokens	88	Incorrect implementation of standard event emission or M manipulation.
Incorrect method invocation	34	The unnamed method rather than the standard methods is invoked.
Lack of event/M modification	2,097	The token contract does not emit the standard event or modify M.
Fee	51	The code of fee charging is implemented in a standard method, or in a non-standard method without proper implementation of standard events.
Token minting	654	The code of token minting is implemented in a standard method, or in a non-standard method without proper implementation of standard events.
Token burning	463	The code of token burning is implemented in a standard method, or in a non-standard method without proper implementation of standard events.
Token purchase	246	An account buys tokens in ETH by invoking a standard method, or a non-standard method without proper implementation of standard events.
Token sell	18	An account sells tokens for ETH by invoking a standard method, or a non-standard method without proper implementation of standard events.
Unit conversion	19	Converting the token into a much smaller basic unit, and the code of unit conversion is implemented in a standard method, or in a non-standard method without proper implementation of standard events.
Account changed	50	The balance of a specified account, rather than the account indicated by standard method interfaces or standard events, is modified.
Amount changed	6	The specified amount of tokens, rather than the amount indicated by standard method interfaces or standard events, are transferred.

6 REASONS OF INCONSISTENT BEHAVIORS

We manually investigate all 2,352 open-source inconsistent tokens to reveal the reasons behind inconsistency. Table 4 lists the 11 major reasons, some of which have several sub-categories. For each reason, we explain it and show the number of inconsistent tokens. Note that one inconsistency can be caused by several reasons. The most number of reasons we find for an inconsistent token is 3, and we find 68 such kind of inconsistent tokens. The figures in "<>" denote the numbers of tokens in the corresponding categories. Note that all inconsistent tokens presented in this section have been deployed in Ethereum and invoked (e.g., purchase/sell/transfer) by accounts.

6.1 Flawed tokens

Flawed tokens implement the standard event emission or M manipulation incorrectly. We found 88 flawed tokens and classified them into four groups. To the best of our knowledge, 50 flawed tokens are unreported. All flawed tokens are listed in http://bit.ly/Tokenscope due to page limit.

(1) *Incorrect implementation of* `Transfer` <24>. We find various errors in implementing the `Transfer` event, such as incorrect accounts, and duplicated events. The consequence is the confusion of users because they cannot know the real token behaviors from the `Transfer` event. Fig. 13 presents a flawed token that sets an incorrect account in `Transfer` (Line 5). Both the semantics of `transferFrom()` (Line 1) and the modification of `M` (Lines 2, 3) indicate that the token sender is *_from*. Therefore, the correct implementation of the `Transfer` event should be "Transfer(_from, _to, _value);" instead of the one on Line 5. As another example, Fig. 14 shows a token that emits the `Transfer` event twice (Lines 2, 6) for each invocation of `transfer()`. A token holder will be confused since the `Transfer` events suggest that $2 \times$ *_value* tokens are transferred by invoking `transfer()`.

```
1 function transferFrom(address _from, address _to, uint256 _value) returns (bool success){
  ......
2   balanceOf[_from] -= _value;
3   balanceOf[_to] += _value;
4   spentAllowance[_from][msg.sender] -= _value;
5   Transfer(msg.sender, _to, _value);}
```

Figure 13: Incorrect implementation of `Transfer` event

```
1 function transfer(address _to, uint256 _value) returns (bool success) {
2     Transfer(msg.sender, _to, _value);
3     if (balances[msg.sender] >= _value && _value > 0) {
4         balances[msg.sender] -= _value;
5         balances[_to] += _value;
6         Transfer(msg.sender, _to, _value);
7         return true;
8     } else { return false; }}
```

Figure 14: A flawed token that emits the `Transfer` event twice

(2)*Informing ETH transfer instead of token transfer* <2>. The event `Transfer` is used to inform token transfer, but we found 2 token contracts that use it to inform ETH transfer (i.e., an account transfers some ETH to another account). Users could be confused because there is no token transfer when they receive the `Transfer` event.

(3) *Incorrect implementation of* `M` *manipulation* <14>. Such flawed tokens will incur serious consequences including financial loss, because the flawed manipulation of `M` may set an incorrect balance to an unexpected account. Fig. 15, Fig. 16 and Fig. 17 present three such flawed tokens. The token contract shown in Fig. 15 sets the balance of *_receiver* to an incorrect value (Line 4) because the balance of *_receiver* rather than that of *msg.sender* should be added. Hence, Line 4 should be "UserBalances[_receiver] = Add(UserBalances[_receiver], _amount);". The token contract shown in Fig. 16 reduces the balance of the token sender twice for each invocation of the standard method `transfer()`. Specifically, the balance is reduced at Line 2, and then the standard method `transferFrom()` is invoked (Line 3), which reduces the balance again. Fig. 17 shows a token containing a subtle flaw. The variable *balanceFrom* is set to the token balance of *_from* (Line 2). Then, the token balance of *_from* is set to the subtraction of *_value* from *balanceFrom* (Line 4). The balance should not be changed if *_from* is *_to*. However, the token balance of *_from* will decrease by *_value* due to the subtle implementation error, if *_from* is *_to*.

```
1 function transfer(address _receiver, uint256 _amount) ...{
2     require(_transferCheck(msg.sender, _receiver, _amount));
3     UserBalances[msg.sender] = Sub(UserBalances[msg.sender], _amount);
4     UserBalances[_receiver] = Add(UserBalances[msg.sender], _amount);
5     Transfer(msg.sender, _receiver, _amount);
6     return true; }
```

Figure 15: A flawed token that sets an incorrect balance

```
1 function transfer(address _toAddress, uint256 _amountOfTokens)...{
  ......
2     tokenBalanceLedger_[_customerAddress] = SafeMath.sub(
        tokenBalanceLedger_[_customerAddress], _amountOfTokens);
3     transferFrom(_customerAddress, _toAddress, _amountOfTokens); ...}
```

Figure 16: A flawed token that reduces token balance twice

```
1 function transferFrom(address _from, address _to, uint256 _value) ... {
  ......
2     uint256 balanceFrom = balances[_from];
  ......
3     balances[_to] = safeAdd(balances[_to], _value);
4     balances[_from] = safeSub(balanceFrom, _value);..... }
```

Figure 17: A flawed token with a subtle error

```
1 function transfer(address to, uint256 value) public returns (bool) {
2     tokens = value * 10 ** decimals;
3     balance[to] = balance[to] + tokens;
4     balance[owner]   = balance[owner] - tokens;
5     emit Transfer(owner, to, tokens); }
```

Figure 18: A flawed token with an integer overflow bug

(4) *Integer overflow* <50>. In EVM, an integer has a maximum value so that an integer overflow happens if an operation results in a value greater than the maximum value, causing the value to wrap-around [61]. A token contract often uses "uint256", a 256-bit unsigned integer which is the longest number supported by EVM [63], to store token balance, and this value will be incorrect if integer overflow happens. Integer overflow has become a major threat to the security of smart contracts, leading to severe consequences (e.g., the market prices of tokens drop, exchange markets suspend token deposits and withdraws) [33, 45, 53]. Fig. 18 presents an inconsistent token having an integer overflow bug (Line 4). When invoking `transfer()` with a large *value*, the result of *balance[owner] - tokens* could be overflowed. Consequently, the token balance of owner increases after subtraction (Line 4).

6.2 Incorrect method invocation

For 34 inconsistent tokens, users attempt to invoke standard interfaces, however, the unnamed method is invoked since the standard methods are not implemented. Note that in EVM the unnamed method will be invoked if the transaction does not specify the invoked method, or the invoked method is not implemented in the contract [63]. Consequently, users may be confused because they intend to call standard methods, rather than the unnamed method. This kind of inconsistency can incur serious security problems, such as token stolen, token frozen (detailed in §7).

6.3 Lack of `Transfer` event and/or `M` modification

Since TokenScope detects inconsistent tokens by comparing B_m, B_e and B_r if $B_m \neq \emptyset$, or comparing B_e and B_r otherwise, the lack of B_e or B_r results in inconsistency. 2,097 inconsistent tokens resulted from this reason, which can be divided into three groups.

(1) $B_e = \emptyset$ <1,405>. The lack of standard event emission hinders the third-party tools from knowing token behaviors. In particular,

users do not know where their tokens come from or go to, even if they could call `balanceOf()` to check their token balances.
(2) $B_r = \emptyset$ <44>. The lack of `M` manipulation indicates that no token transfers happen in practice. Consequently, users may be confused because the `Transfer` event informs token transfers. KYC Casper Token reported by a recent news belongs to this category [64].
(3) $B_e = B_r = \emptyset$ <833>. We find 833 inconsistent tokens that neither manipulate `M` nor emit the `Transfer` event, when executing standard methods. As a result, the third-party tools (e.g., exchange markets) that detect token behaviors by monitoring the invocation of standard interfaces will incorrectly think that token transfer happens. This issue will cause "fake deposit" [40]. Fig. 19 shows an example detected by `TokenScope`. Line 7 will be executed if the comparison in Line 2 returns false. In this case, the standard method `transfer()` executes without modifying `M` and emitting the standard event.

```
1  function transfer(address _to, uint256 _value) public validAddress(_to) ... {
2    if(balanceOf[msg.sender] >= _value && _value > 0){
3      balanceOf[msg.sender] = sub(balanceOf[msg.sender], _value);
4      balanceOf[_to] = add(balanceOf[_to], _value);
5      Transfer(msg.sender, _to, _value);
6      return true;}
7    else{return false;}}
```

Figure 19: An inconsistency because $B_e = B_r = \emptyset$

6.4 Fee

A token contract can charge fee from any token holder, and the fee is sent to the token contract, the token creator, or any account specified by the token creator. An inconsistent behavior happens if the code of fee charging is written (1) in a standard method, or (2) in a non-standard method without proper implementation of standard events. 51 inconsistent tokens are due to this reason, and Fig. 20 presents one. A token holder, *msg.sender* intends to transfer *_value* tokens to a token holder *_to* by invoking `transfer()` (Line 1). The balance of *msg.sender* is decreased by *_value* (Line 3), however, only $\_value - \_value \times fee \,/\, 10{,}000$ tokens are transferred to *_to* (Line 4). The remaining tokens are charged as fee and sent to another token holder, *_feeWallet* (Line 5). The token emits two `Transfer` events that faithfully reflect token behaviors (Lines 6, 7). Our approach detects the inconsistency because B_m is different from B_e and B_r. Note that the account who invokes `transfer()` may not intend to transfer money to *_feeWallet*.

```
1  function transfer(address _to, uint256 _value) public returns (bool){
2    require(_to != address(0));
3    balances[msg.sender] = balances[msg.sender].sub(_value);
4    balances[_to] = balances[_to].add(_value.sub(_value * fee / 10000));
5    balances[_feeWallet] = balances[feeWallet].add(_value * fee / 10000);
6    Transfer(msg.sender, _to, (_value.sub(_value * fee / 10000)));
7    Transfer(msg.sender, feeWallet, (_value * fee / 10000));
8    return true;}
```

Figure 20: The inconsistency incurred by charging fee

6.5 Token minting

Token minting means increasing the total amount of tokens in circulation. If the code of token minting is written (1) in a standard method, or (2) in a non-standard method without proper implementation of standard events, an inconsistency happens. We detect token minting according to B_r. More precisely, for every trace, we sum the token changes of all token holders whose balances are increased. Similarly, we sum the (absolute value) token changes of all token holders whose balances are decreased. Then, we check whether the first summation is larger than the second one. If so, we think that token minting happens. Please reconsider the contract in Fig. 2, the balances of *_to* and *hacker* are increased by *_value* and *fee*, respectively and then the first summation is $\_value + fee$. The balance of *msg.sender* is decreased by $\_value - fee$. Hence, the first summation is equal to the second, and thus no token minting happens in this contract. 654 inconsistent tokens are due to token minting. We classify those 654 tokens into five minor categories, and the figures in "<>" stand for the numbers of inconsistent tokens belonging to the sub-categories.
(1) Reward <635>. A token contract can implement various strategies to reward users with some amount of tokens. For example, a token rewards the accounts who produce the block or call the token contract for the first time.
(2) Subsidy <2>. Ethereum requires transaction senders to pay transaction fee in ETH to prevent resource abusing [6]. To attract users to invoke token contracts, many token contracts send users some amount of tokens as the subsidy for transaction fee.
(3) Donation <4>. We find that 4 token contracts donate some amount of tokens to specified accounts for each invocation of the token contracts.
(4) Token migration <8>. Token developers will deploy a new version of token contract on the blockchain to substitute the old version for some reasons (e.g., fix bugs). After the deployment of the new contract, the new contract should migrate some data from the old contract, e.g., the addresses of token holders, the amount of tokens possessed by token holders. Without migration, users' tokens will be lost.
(5) Unlocking <9>. The founders of a token contract can lock their proportions of tokens to increase the confidence of other users. The locked tokens cannot be circulated because they are not recorded in `M`. In other words, they can be neither sold nor transferred to other users. The locked tokens will be unlocked when some conditions are met, e.g., the locking period is expired. Token unlocking results in token minting, because the unlocked tokens will be added in `M`.

Fig. 21 presents a deployed inconsistent token due to both subsidy and donation. A token holder intends to transfer *tokens* tokens to the holder *to* by invoking `transfer()` (Line 1), and the `Transfer` event is accordant with `transfer()` (Line 6). The token contract sends 5,000 tokens to *msg.sender* as subsidy (Line 3), and it donates 5,000 tokens to an account, *donation* (Line 5). The contract emits a non-standard event Donation (Line 7), however, the semantics of Donation is unclear. Consequently, the transaction sender may not know the occurrence of token donation. By leveraging a wallet, the sender knows that the balance of *msg.sender* decreases by *tokens* - 5,000. Hence, the transaction sender could be confused because the sender intends to send *tokens* tokens to *to* by invoking `transfer()`.

```
1  function transfer(address to, uint tokens) public returns (bool success){
2    address donation = donationsTo[msg.sender];
3    balances[msg.sender] = (balances[msg.sender].sub(tokens)).add(5000);
4    balances[to] = balances[to].add(tokens);
5    balances[donation] = balances[donation].add(5000);
6    emit Transfer(msg.sender, to, tokens);
7    emit Donation(donation);
8    return true;}
```

Figure 21: The inconsistency due to subsidy and donation

6.6 Token burning

Token burning means decreasing the total amount of tokens in circulation. We detect an inconsistency if the code of token burning is written (1) in a standard method, or (2) in a non-standard method without proper implementation of standard events. We detect token burning using a similar approach for detecting token minting, except that token burning happens if the first summation is smaller than the second one. 463 inconsistent tokens are due to token burning. We classify them into two sub-categories.
(1) Wear <9>. Some token contracts charge fee, but the fee is not sent to any account so that the fee disappears during token transfer.
(2) Reclaim <454>. Some token contracts burn tokens when some tokens are sent to the address 0, the addresses of token contracts, the addresses of token creators or any accounts specified by token creators. In such case, token contracts intend to reclaim those tokens.

Fig. 22 presents a deployed inconsistent token due to reclaim. The token contract first calls super.`transferFrom()` to transfer *_value* tokens from the account *_from* to the account *_to* (Line 2). Then, the token contract checks whether the tokens are sent to the token contract (Line 3). If so, the transferred tokens are burned (Line 4). Although the token contract emits a non-standard event, Destruction (Line 6), the semantics of this event are unclear and thus the transaction sender may not know token burning.

```
1  function transferFrom(address _from, address _to, uint256 _value)... {
2      assert(super.transferFrom(_from, _to, _value));
3      if (_to == address(this)) {
4          balanceOf[_to] -= _value;
5          totalSupply -= _value;
6          Destruction(_value);
       }
7      return true;
   }
```

Figure 22: The inconsistency incurred by reclaim

6.7 Token purchase

Some token contracts allow automatic token purchase without the interference of exchange markets. These token contracts send some amount of tokens to the accounts who send ETH to the token contracts according to the exchange rate implemented in the token contracts. An inconsistent behavior occurs if the code of token purchase is written (1) in a standard method, or (2) in a non-standard method without proper implementation of standard events. 246 inconsistent tokens are due to token purchase. Fig. 23 presents an inconsistent token due to token purchase. The transaction sender pays *msg.value* ETH to purchase *qiuAmount* tokens (Line 2). The token contract sends *qiuAmount* tokens to *msg.sender* (Lines 4, 5), and then emits a non-standard event (Line 6). The consequence of the inconsistency is that although users can check their token balances by invoking the standard method `balanceOf()`, they may not know why their balances increase because the semantics of the non-standard events are unclear.

6.8 Token sell

Some token contracts check whether the ETH possessed by an account is smaller than a threshold. If so, token contracts charge some amount of tokens from the account and send some amount of ETH to that account according to the exchange rate implemented in the token contracts. Token sell will incur inconsistency if the code of token sell is written (1) in a standard method, or (2) in a non-standard method without proper implementation of standard events. 18 inconsistent tokens are due to token sell. Fig. 24 shows a deployed inconsistent token due to token sell. If the ETH possessed by the token receiver *_to* is less than a threshold, *minBalanceForAccounts* (Line 2), the amount of tokens, *amountinBoss* is computed according to the exchange rate, *sellPrice* (Line 3). Then, *amountinBoss* tokens are transferred to the account specified by the token creator (Line 4), and the account *_to* receives *amountinBoss / sellPrice* ETH (Line 5). Such token sell behavior may be troublesome, since any account *A* could convert the tokens of another account *B* into ETH by transferring some tokens to *B*.

```
1  function exchangeForQIU() payable public returns (bool){
2      uint qiuAmount = msg.value * eth2qiuRate / 1000000000000000000;
3      require(qiuAmount <= balances[this]);
4      balances[this] = balances[this].sub(qiuAmount);
5      balances[msg.sender] = balances[msg.sender].add(qiuAmount);
6      ExchangeForQIU(this, msg.sender, qiuAmount, msg.value);
7      return true;
   }
```

Figure 23: The inconsistency incurred by token purchase

```
1  function _transfer(address _from, address, _to, uint _value) internal{
   ......
2      if(_to.balance < minBalanceForAccounts){
3          uint256 amountinBoss = (minBalanceForAccounts - _to.balance) * sellPrice;
4          _transfer(_to, owner, amountinBoss);
5          _to.transfer(amountinBoss / sellPrice);}
6      Transfer(_from, _to, _value);}
```

Figure 24: The inconsistency incurred by token sell

6.9 Unit conversion

Some token contracts specify a basic unit of tokens, which is much smaller than one token. Unit conversion will lead to inconsistency if the code of unit conversion is written (1) in a standard method, or (2) in a non-standard method without proper implementation of standard events. We detect 19 inconsistent tokens due to such reason. Fig. 25 presents a real case. The `Transfer` event informs that the user *userAddress[myid]* receives *no_of_token* tokens (Line 4). However, the tokens are converted into the basic unit before token transfer (Lines 2, 3). The basic unit is $1/10^{10}$ of one token. Consequently, the user may be confused because the user will find that the token balance is significantly larger than the amount informed by the `Transfer` event, when checking the balance by invoking `balanceOf()` (Lines 5, 6).

```
1  function __callback(bytes32 myid, string result){
   ......
2      balances[owner] -= (no_of_token * 10000000000);
3      balances[userAddress[myid]] += (no_of_token * 10000000000);
4      Transfer(owner, userAddress[myid], no_of_token);
   ......}
5  function balanceOf(address sender) constant returns (uint256 balance) {
6      return balances[sender];}
```

Figure 25: The inconsistency incurred by unit conversion

6.10 Account changed

We find 50 token contracts that change the accounts to send or receive tokens instead of using the account specified by the standard interfaces or standard events. Fig. 26 presents a real case due to this reason. The transaction sender intends to transfer some tokens to

the account *_to* by invoking `transfer()` (Line 1), but *_to* is changed to another account (Line 5) when some conditions are satisfied (Lines 3, 4). Consequently, the transaction sender may feel upset since tokens are sent to a different account rather than the intended one.

```
1  function transfer(address _to, uint256 _value) onlyPayloadSize(2 * 32){
   ......
2     if(_to == deposit_address){ ......}
3     else{
4        if(isLeading4FF(_to)){
         ......
5           _to = shopStoreAddress[uint(storeid)];
         ......}}
6     Transfer(msg.sender, _to, _value);}
```

Figure 26: The inconsistency incurred by account specifying

6.11 Amount changed

Some token contracts change the amount of transferred tokens rather than the amount indicated by standard interfaces or standard events before token transfers. However, users cannot know the change by monitoring the invocation of standard method interfaces. 6 inconsistent tokens are due to amount specifying. Fig. 27 shows a real case. The transaction sender intends to transfer *_value* tokens (Line 1). However, the real transferred amount is restricted to *_value - maxGoalInICO* (Line 6) if some conditions are satisfied (Lines 2, 5). Consequently, the transaction sender may be confused since the real transferred amount is less than the intended amount.

```
1  function transferFrom(address _from, address _to, uint256 _value) ... {
   ......
2     if (now < startTime){
3        if(_value < maxGoalInICO ) {
4           tokensSoldToInvestors = safeAdd(tokensSoldToInvestors, _value);
5        } else {
6           _value = safeSub(_value, maxGoalInICO);}}
      .....
7     Transfer(_from, _to, _value);
8     return true;}
```

Figure 27: The inconsistency incurred by amount specifying

We also show the cumulative distribution function plots of open-source inconsistent tokens and flawed tokens in Fig. 28. Each $\times$ (x, y)/$\circ$ (x, y) indicates that there are y inconsistent/flawed tokens, and there are no more than x external transactions trigger/exploit the inconsistencies/flaws. For about 19% $((2,352-1,908)/2,352)$ of inconsistent tokens and about 10% $((88-79)/88)$ of flawed tokens, there are at least 100 (i.e., more than 99) external transactions that trigger/exploit the inconsistencies/flaws. That is, many inconsistent tokens executed inconsistent behaviors frequently, and many flawed tokens have been exploited frequently. For example, the inconsistent behaviors of IdleEth have been triggered by the most number of external transactions (i.e., 269,204), and the HYDRO token is the flawed token that has been exploited by the most number of external transactions (i.e., 15,032) (detailed in §7).

7 CASE STUDIES

This section presents case studies of six inconsistent tokens: HYDRO, SMT, ZXBT, GTN, Tablow Club, and MCRT. HYDRO has an implementation flaw in `transferFrom()`. SMT contains an integer overflow bug. The other four inconsistent tokens are due to incorrect method invocation. We find that all of them have been attacked

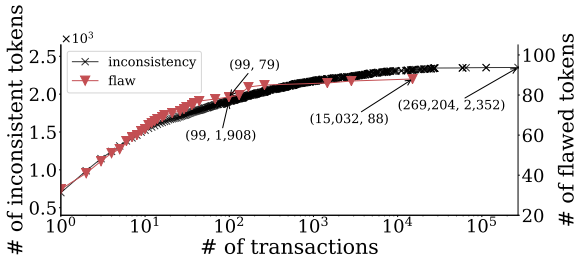

Figure 28: CDFs of open-source inconsistent/flawed tokens

according to our trace analysis, leading to serious consequences, including incorrect token balance, token frozen, or token stolen.

GTN. 26,097 external transactions have been sent to the HYDRO token contract, and 12,705 accounts possess HYDRO Token. We show below how an attacker can steal HYDRO tokens from an exchange market. The standard method `transferFrom()` (Fig. 29) contains a bug resulting in inconsistency. The token behavior of the standard method is that *_from* transfers *_value* tokens to *_to*. However, the real token behavior is that the account who invokes `transferFrom()` (*msg.sender*) transfers the tokens to *_to*. (Lines 2, 5 - 7).

```
1  function transferFrom(address _from, address _to, uint256 _value)...{
   ......
2     _transfer(msg.sender, _to, _value);
3     Transfer(msg.sender, _to, _value);
4     return true;}
5  function _transfer(address _from, address _to, uint _value) internal {
6     balances[_from] -= _value;
7     balances[_to] += _value;}
```

Figure 29: Code snippet of HYDRO token.

```
1  function depositToken(address token, uint amount) {
2     if (token == 0) throw;
3     if (!Token(token).transferFrom(msg.sender, this, amount)) throw;
4     tokens[token][msg.sender] = safeAdd(tokens[token][msg.sender], amount);
5     Deposit(token, msg.sender, amount, tokens[token][msg.sender]); }
6  function withdrawToken(address token, uint amount) {
7     if(token == 0) throw;
8     if(tokens[token][msg.sender] < amount) throw;
9     tokens[token][msg.sender] = safeSub(tokens[token][msg.sender], amount);
10    if (!Token(token).transfer(msg.sender, amount)) throw;
11    Withdraw(token, msg.sender, amount, tokens[token][msg.sender]); }
```

Figure 30: Code snippet of Etherdelta_2

We find 15,032 invocations of `transferFrom()` from another smart contract, EtherDelta_2 which belongs to EtherDelta [9]. EtherDelta is a popular exchange market, and we observe more than 10 million external transactions sent to EtherDelta_2. Users can deposit and withdraw various kinds of tokens by invoking depositToken() and withdrawToken() in EtherDelta_2, receptively (Fig. 30). The kind of token to be deposited or withdrawn is specified by its address, and the amount is specified by *amount*. depositToken() invokes `transferFrom()` in the HYDRO token contract to transfer *amount* tokens from the account who invokes depositToken() (*msg.sender*) to the EtherDelta_2 contract (*this*) (Line 3). Due to the flawed implementation of the HYDRO token contract (Fig. 29), the real token behavior is that the EtherDelta_2 contract (rather than the account who deposits the HYDRO tokens) transfers *amount* tokens to the EtherDelta_2 contract. We find 23 accounts who sent 15,032 transactions to invoke depositToken() of the EtherDelta_2 contract which in turn call `transferFrom()` of the HYDRO token contract. Those 15,032 transactions deposit more than 2.6 billion HYDRO tokens.

To withdraw tokens, a user invokes withdrawToken() to transfer *amount* tokens from the EtherDelta_2 contract to the user. An

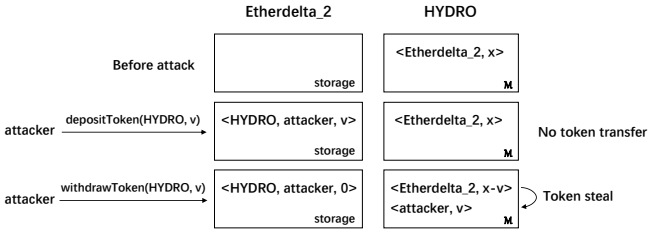

Figure 31: The process of stealing HYDRO.

attacker steals HYDRO tokens from the EtherDelta_2 contract (i.e., the EtherDelta exchange market) by first invoking depositToken() and then invoking withdrawToken(). We found that more than 2.5 million HYDRO tokens were stolen by 25 external transactions sent from 11 accounts. Fig. 31 shows the attack process. We assume that the attacker does not hold HYDRO tokens and EtherDelta_2 holds x HYDRO tokens before attacks. The `M` of HYDRO contains an item corresponding to EtherDelta_2. An attacker invokes depositToken() with parameters HYDRO and v to transfer v HYDRO tokens from the attacker to EtherDelta_2. EtherDelta_2 records such token deposit behavior in its storage. Due to the implementation flaw in `transferFrom()` as shown in Fig. 29, EtherDelta_2 instead of the attacker transfers v HYDRO tokens to EtherDelta_2, and thus no token transfer happens. After that, the attacker invokes withdrawToken() to withdraw v HYDRO tokens from EtherDelta_2. EtherDelta_2 updates the corresponding record and the HYDRO contract transfers v tokens from EtherDelta_2 to the attacker. Consequently, the attacker steals v HYDRO tokens from EtherDelta_2.

SMT. SMT has an integer overflow bug and was deployed to the blockchain on Dec. 09, 2017. The first attack exploiting the bug happened on Apr. 24, 2018 and the first report about this attack was published on Apr. 25, 2018 [44]. Fig. 32 shows the method transferProxy() that contains the integer overflow bug. The token balance of *_from* will be decreased by *_value* + *_feeSmt* (Line 7), and Line 2 checks whether *_from* possesses sufficient tokens. However, the summation *_feeSmt* + *_value* can be overflowed providing a big *_feeSmt* or a big *_value*. Consequently, the check can be passed because the summation is a small value due to integer overflow, and the account *_to* or *msg.sender* will receive a great amount of tokens (Lines 3, 5).

```
1  function transferProxy(address _from, address _to,
     uint256 _value, uint256 _feeSmt... {
2    if(balances[_from] < _feeSmt + _value) revert();
3    balances[_to] += _value;
4    Transfer(_from, _to, _value);
5    balances[msg.sender] += _feeSmt;
6    Transfer(_from, msg.sender, _feeSmt);
7    balances[_from] -= _value + _feeSmt;...}
```

Figure 32: Code snippet of SMT token.

TokenScope detects a transaction (transaction hash: 0x1abab4c8db9a30e703114528e31dee129a3a758f7f8abc3b6494aad3d304e43f) which exploited the vulnerability. In this attack, *_from* and *_to* are the same account, and *_msg.sender* is a different account. The balances of the two accounts before the attack are 0. The attacking transaction sets *_value* and *_feeSmt* to two big integers, 0x8fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff and 0x7000000000000000000000000000000000000000000000000000000000000001, respectively. Hence, the summation of *_value* and *_feeSmt* is 0 due to integer overflow. After the attack, the balance of *_from* (or *_to*) becomes 0x8fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff, and the balance of *_msg.sender* is 0x7000000000000000000000000000000000000000000000000000000000000001.

ZXBT. Its token contract does not implement the standard method `transfer()`. Consequently, when a user attempts to invoke `transfer()`, the unnamed method will be invoked instead. ZXBT can be traded on EtherDelta [9], which deploys EtherDelta_2 to manage this token. Due to the implementation issue of ZXBT, user's token will be frozen. That is, the user can neither withdraw nor sell ZXBT. Fig. 30 shows the code of EtherDelta_2. A user invokes depositToken() (Line 1) to deposit ZXBT to EtherDelta_2, and then invokes withdrawToken() (Line 6) to withdraw ZXBT from EtherDelta_2. Since the implementation of `transferFrom()` in this token contract is correct, the user successfully deposits ZXBT to EtherDelta_2. However, the user cannot withdraw ZXBT because EtherDelta_2 invokes the unnamed method, which does not transfer tokens, rather than the `transfer()` (Line 10). We found that 7,115,006 ZXBT is frozen which was worth about 3,000 USD when ZXBT was deposited.

GTN. Its contract does not implement the standard method `transferFrom()`. Therefore, when a user attempts to invoke its `transferFrom()`, the unnamed method will be invoked. We find that the GTN token can also be traded on EtherDelta. Consequently, the method depositToken() does not transfer GTN to Etherdelta_2 (shown in Fig. 30) because it invokes the unnamed method rather than `transferFrom()` (Line 3). However, Etherdelta_2 is not aware of the implementation issue in the token contract of GTN, and hence it mistakenly records that a user deposits GTN to Etherdelta_2 (Line 4). By invoking withdrawToken(), a user can withdraw GTN (Line 10), although the user does not deposit GTN. Hence, an attacker can exploit this implementation flaw to steal GTN from Etherdelta_2. We observe that two accounts successfully exploited the issue to steal 3,000,000 GTN from Etherdelta_2. Besides GTN, attackers stole Tablow Club and MCRT from Etherdelta_2 due to the same reason.

8 DISCUSSION

We discuss the limitations of TokenScope and potential solutions.

Other token standards. We focus on 2 standard interfaces `transfer()` and `transferFrom()` and 1 standard event `Transfer` defined in ERC-20. Fortunately, to be compatible with ERC-20 or at least avoid conflicting with ERC-20, other standards typically support `transfer()`, `transferFrom()` and `Transfer` defined in ERC-20. We will extend TokenScope to support other standard methods and events of ERC-20 as well as other token standards in future work.

Deliberate evasion. Smart contracts can deliberately evade the detection of TokenScope by using other data structures instead of those recognized by TokenScope. However, using a deliberately crafted data structure for `M` may lead to a more complicated implementation of the token contract, and thus increase the cost (i.e., gas) of deploying and invoking the token. We will investigate how to automatically infer the data structures and accessing patterns of `M` in future work. Such automatic inference is possible because the two storage locations (i.e., the space to store balances) derived from the two addresses (i.e., the token sender and the token receiver) will be written when token transfers. By monitoring the access to the two storage locations, we can learn the access pattern and then infer the data structure. In particular, we will first conduct program slice to extract the operations that are related to the storage

modifications, and then identify the access patterns from the slices. Another possible evasion approach to disperse a typical token behavior (e.g., token transfer) into several methods and then perform the token behavior by sending several external transactions to invoke those methods (i.e., one external transaction just triggers part of a typical token behavior). Thus, TokenScope may produce false positives because it detects inconsistency per trace which is corresponding to one external transaction. We will improve TokenScope by conducting cross-transaction analysis in future work.

9 RELATED WORK

Token analysis. Somin et al. identify token transfers by parsing the Transfer event [56], which does not necessarily reflect real token behaviors. The differences between our work with Fröwis et al.'s work [20] are described in §1. SECBIT maintains a collection of buggy ERC-20 tokens [52], but it mainly focuses on common token problems (e.g., weak access control) rather than inconsistency, and we find that about 95.7% of inconsistent tokens detected by TokenScope are not disclosed in its list.

Vulnerability discovery. SmartCheck detects 21 kinds of bugs in Solidity source code by searching for bug patterns [57]. It cannot be easily extended to detect inconsistency in EVM bytecode because (1) it needs the source code of smart contracts; and (2) the detection of inconsistency needs to understand program semantics but pattern searching does not support it. EtherTrust [24] detects two kinds of security bugs based on formal semantics of EVM bytecode [25]. Vandal decompiles EVM bytecode into semantic logic relations and detects five kinds of security problems which are expressed by logic specifications [3]. MadMax detects security problems using Vandal for bytecode decompilation [23]. Sereum builds on-line taint analysis into EVM to protect smart contracts from reentrancy attacks [48].

teEther produces transactions by symbolic execution (SE) to find and exploit the vulnerabilities of a smart contract [35]. Osiris combines SE and taint analysis to discover integer overflow bugs in EVM bytecode [58]. EthRacer integrates fuzzing of event sequences and SE to check whether a contract produces different outputs by re-ordering event sequences [34]. sCompile applies SE to critical paths which involve money transfer, and leaves the other paths unexplored [4]. Huang detects security problems in EVM bytecode via deep learning [29]. Parizi et al. study four tools about their capabilities to discover security bugs [43], and find that SmartCheck achieves the highest accuracy [43]. ContractFuzzer applies fuzzing to discover seven kinds of security problems [30]. Grossman et al. detect the reentrancy bug by focusing on the callback nature of smart contracts [26]. In summary, these techniques focus on vulnerability discovery, especially security vulnerabilities rather than inconsistent token behaviors violating ERC-20.

General analysis platforms. K framework [49, 50] is based on the formal semantic of KEVM [27] and is possible to detect the inconsistency happened in standard methods because the semantics of standard methods interfaces are known. However, K framework is not fully automated. For example, to apply it for checking inconsistency, users have to provide the identity of M in the specification. Differently, our approach locates M automatically. Second, K framework requires the developers of token contracts to write specifications for analyzing non-standard methods since their semantics are unknown. Differently, our approach can automatically detect the inconsistency in non-standard methods. Chatterjee et al. propose to infer the lower bound and upper bound of a variable in their proposed language [5]. The contract is considered as an incorrect one if the expected value does not fall into the interval [lower bound, upper bound] [5]. However, their method may suffer from false negatives, e.g., an incorrect value can also fall into the interval. Moreover, inference of the expected value is non-trivial because it requires a deep understanding of the analyzed contract and the semantics of EVM operations. A few works propose formal semantics of EVM and EVM bytecode [1, 25, 28, 31] to facilitate correctness verification, but they do not provide an automated verifier.

Securify [59] decompiles EVM bytecode, and uses a domain-specific language (DSL) to express several security properties. Then, it analyzes smart contracts to check those security properties. Securify does not recover the types of variables during decompilation, and hence it cannot locate M in a token contract. Besides, whether DSL can express inconsistency is unknown. Zeus [32] is a security verifier that needs the source code of smart contracts. It converts the specification written in XACML [55] into checking code, and then inserts the checking code into the source code of smart contracts. After that, Zeus translates the modified source code into an intermediate language, and then applies abstract interpretation and symbolic model checking to check security properties. The applicability of Zeus is restricted since open-source smart contracts only account for less than 1% of all contracts [19]. In contrast, our approach directly processes EVM bytecode. Moreover, whether XACML is able to express inconsistency is unknown.

10 CONCLUSION

Inconsistent behaviors can mislead users and cause severe financial loss, such as money frozen and money stolen. We propose a novel approach and develop a new tool named TokenScope to automatically detect inconsistent behaviors resulted from tokens deployed in Ethereum by comparing the information from three different sources, including the manipulations of core data structures, the actions indicated by standard interfaces, and the behaviors suggested by standard events. Applying TokenScope to inspect all transactions sent to all deployed tokens, we find 3,259,001 transactions which trigger inconsistent behaviors, and 7,472 inconsistent tokens with a very high precision. The investigation of all open-source inconsistent tokens reveals 11 major reasons behind the inconsistency, including 50 unreported flawed tokens.

ACKNOWLEDGEMENT

Ting Chen is partially supported by National Natural Science Foundation of China (61872057) and National Key R&D Program of China (2018YFB0804100). Ting Wang is partially supported by the National Science Foundation under Grant No. 1718787 and 1846151.

REFERENCES

[1] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *ACM SIGPLAN International Conference on Certified Programs and Proofs*.

[2] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nkhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal verification of smart contracts: Short paper. In *ACM Workshop on Programming Languages and Analysis for Security*.

[3] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security

Analysis Framework for Smart Contracts. https://arxiv.org/pdf/1809.03981.pdf. (2018).
[4] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, and Zijiang Yang. 2018. sCompile: Critical Path Identification and Analysis for Smart Contracts. https://arxiv.org/pdf/1808.00624.pdf. (2018).
[5] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. 2018. Quantitative Analysis of Smart Contracts. In *European Symposium on Programming*.
[6] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *International Conference on Software Analysis, Evolution and Reengineering*.
[7] Curvegrid. 2018. toy-block-explorer. https://github.com/curvegrid/toy-block-explorer. (2018).
[8] enkrypt. 2018. EthVM: Open Source Ethereum Blockchain Explorer. https://github.com/enKryptIO/ethvm. (2018).
[9] EtherDelta. 2018. EtherDelta. https://etherdelta.com/. (2018).
[10] Ethereum. 2017. ERC223 token standard. https://github.com/ethereum/EIPs/issues/223. (2017).
[11] Ethereum. 2017. Management APIs. https://github.com/ethereum/go-ethereum/wiki/Management-APIs. (2017).
[12] Ethereum. 2017. Token Standard Extension for Increasing & Decreasing Supply. https://github.com/ethereum/EIPs/pull/621. (2017).
[13] Ethereum. 2018. ETCExplorer. https://github.com/ethereumclassic/explorer. (2018).
[14] Ethereum. 2018. Etherscan — The Ethereum Block Explorer. https://etherscan.io/. (2018).
[15] EtherEx. 2018. EthEx: Decentralized exchange built on Ethereum. https://github.com/etherex/etherex. (2018).
[16] Etherscan. 2018. Token Tracker. https://etherscan.io/tokens. (2018).
[17] Etherscan. 2019. Decentralized Exchange Order Tracker. https://etherscan.io/dextracker. (2019).
[18] Etherwall. 2018. Etherwall: The first Ethereum desktop wallet. https://www.etherwall.com/. (2018).
[19] Michael Fröwis and Rainer Böhme. 2017. In Code We Trust? Measuring the Control Flow Immutability of All Smart Contracts Deployed on Ethereum. In *International Workshops on Data Privacy Management, Cryptocurrencies and Blockchain Technology*.
[20] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. 2018. Detecting Token Systems on Ethereum. https://arxiv.org/pdf/1811.11645.pdf. (2018).
[21] FunFairTech. 2017. Funfair token contract update. https://www.reddit.com/r/FunfairTech/comments/6nadvm/funfair_token_contract_update/. (2017).
[22] Google. 2019. Ethereum ETL. https://github.com/blockchain-etl/ethereum-etl. (2019).
[23] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. In *ACM international conference on Object-oriented Programming, Systems, Languages, and Applications*.
[24] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. EtherTrust: Sound Static Analysis of Ethereum bytecode. https://www.netidee.at/sites/default/files/2018-07/staticanalysis.pdf. (2018).
[25] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum smart contracts. In *International Conference on Principles of Security and Trust*.
[26] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. In *ACM SIGPLAN Symposium on Principles of Programming Languages*.
[27] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, , and Grigore Rosu. 2017. KEVM: A Complete Semantics of the Ethereum Virtual Machine. https://www.ideals.illinois.edu/bitstream/handle/2142/97207/hildenbrandt-saxena-zhu-rodrigues-guth-daian-rosu-2017-tr.pdf?sequence=2. (2017).
[28] Yoichi Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*.
[29] TonTon Hsien-De Huang. 2018. Hunting the Ethereum Smart Contract: Color-inspired Inspection of Potential Attacks. https://arxiv.org/pdf/1807.01868.pdf. (2018).
[30] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *ACM/IEEE International Conference on Automated Software Engineering*.
[31] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2018. Executable Operational Semantics of Solidity. https://arxiv.org/pdf/1804.01295.pdf. (2018).
[32] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing safety of smart contracts. In *The Network and Distributed System Security Symposium*.
[33] Kaustav. 2018. The Effects of the ERC20 Batch Overflow Bug. https://globalcoinreport.com/the-effects-of-the-erc20-batch-overflow-bug/. (2018).
[34] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2018. Exploiting The Laws of Order in Smart Contracts. https://arxiv.org/pdf/1810.11605.pdf. (2018).
[35] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *USENIX Security Symposium*.
[36] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *ACM SIGSAC Conference on Computer and Communications Security*.
[37] METAMASK. 2018. METAMASK — Brings Ethereum to your browser. https://metamask.io/. (2018).
[38] MyEtherWallet. 2018. MyEtherWallet. https://www.myetherwallet.com/. (2018).
[39] Mythril. 2018. Mythril Platform enables a secure and thriving ecosystem of Ethereum dapps & smarts contracts. https://mythril.ai/. (2018).
[40] OKCoin. 2018. OKEx Safe from USDT "Fake Deposit" Issue. https://support.okex.com/hc/en-us/articles/360006305532-OKEx-Safe-from-USDT-Fake-Deposit-Issue. (2018).
[41] openANX. 2017. openANX: Decentralised Exchange Token Sale Smart Contract. https://github.com/openanx/OpenANXToken. (2017).
[42] OpenZeppelin. 2019. SafeMath Library. https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol. (2019).
[43] Reza M. Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains. In *Annual International Conference on Computer Science and Software Engineering*.
[44] peckchield. 2018. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://blog.peckshield.com/2018/04/25/proxyOverflow/. (2018).
[45] PeckShield. 2018. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://blog.peckshield.com/2018/04/25/proxyOverflow/. (2018).
[46] Plutocracy. 2019. Krown whitepaper. https://plutocracy.co/resources/pdf/Plutocracy_Whitepaper.pdf. (2019).
[47] POA. 2018. BlockScout, Blockchain Explorer for inspecting and analyzing EVM Chains. https://github.com/poanetwork/blockscout. (2018).
[48] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *The Network and Distributed System Security Symposium*.
[49] Grigore Rosu. 2017. *K: A Semantic Framework for Programming Languages and Formal Analysis Tools*.
[50] Grigore Rosu. 2018. Formal Design, Implementation and Verification of Blockchain Languages (Invited Talk). In *Leibniz International Proceedings in Informatics*.
[51] Amitabha Sanyal, Bageshri Sathe, and Uday Khedker. 2009. *Data flow analysis: theory and practice. CRC Press, 2009.* CRC Press.
[52] SECBIT. 2018. bad_tokens.all.csv. https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/bad_tokens.all.csv. (2018).
[53] Oguz Serdar. 2018. Ethereum bug causes integer overflow in numerous ERC20 smart contracts [Update]. https://thenextweb.com/hardfork/2018/04/25/ethereum-smart-contract-integer-overflow/. (2018).
[54] Matthew De Silva. 2017. Ethereum Improvement Proposal 20 Finalized, Formally Establishes ERC20 Standard. https://www.ethnews.com/ethereum-improvement-proposal-20-finalized-formally-establishes-erc20-standard. (2017).
[55] Remon Sinnema. 2013. eXtensible Access Control Markup Language (XACML) XML Media Type. https://tools.ietf.org/html/rfc7061. (2013).
[56] Shahar Somin, Goren Gordon, and Yaniv Altshuler. 2018. Network Analysis of ERC20 Tokens Trading on Ethereum Blockchain. In *International Conference on Complex Systems*.
[57] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2016. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain*.
[58] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Annual Computer Security Applications Conference*.
[59] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *ACM SIGSAC Conference on Computer and Communications Security*.
[60] Haijun Wang, Yi Li, Shangwei Lin, Lei May, and Yang Liu. 2019. VULTRON: Catching Vulnerable Smart Contracts Once and for All. In *International Conference on Software Engineering – NIER*.
[61] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *The Network and Distributed System Security Symposium*.
[62] WIKI. 2018. ERC20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md. (2018).
[63] Gavin Wood. 2018. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf. (2018).
[64] ZeusTrade. 2018. Topic: there was a coin out of my wallet that I did not even get what it is. https://bitcointalk.org/index.php?topic=5023796.0. (2018).