RESEARCH-ARTICLE

TokenAware: Accurate and Efficient Bookkeeping Recognition for Token Smart Contracts

ZHEYUAN HE, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

SHUWEI SONG, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

YANG BAI, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

XIAPU LUO, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

TING CHEN, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

WENSHENG ZHANG, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

View all

Published: 13 February 2023
Online AM: 29 August 2022
Accepted: 22 July 2022
Revised: 14 July 2022
Received: 15 November 2021

TokenAware: Accurate and Efficient Bookkeeping Recognition for Token Smart Contracts

ZHEYUAN HE, SHUWEI SONG, and YANG BAI, University of Electronic Science and Technology of China, China
XIAPU LUO, The Hong Kong Polytechnic University, China
TING CHEN, WENSHENG ZHANG, PENG HE, and HONGWEI LI, University of Electronic Science and Technology of China, China
XIAODONG LIN, University of Guelph, Canada
XIAOSONG ZHANG, University of Electronic Science and Technology of China, China

Tokens have become an essential part of blockchain ecosystem, so recognizing token transfer behaviors is crucial for applications depending on blockchain. Unfortunately, existing solutions cannot recognize token transfer behaviors accurately and efficiently because of their incomplete patterns and inefficient designs. This work proposes `TokenAware`, a novel online system for recognizing token transfer behaviors. To improve accuracy, `TokenAware` infers token transfer behaviors from modifications of internal bookkeeping of a token smart contract for recording the information of token holders (e.g., their addresses and shares). However, recognizing bookkeeping is challenging, because smart contract bytecode does not contain type information. `TokenAware` overcomes the challenge by first learning the instruction sequences for locating basic types and then deriving the instruction sequences for locating sophisticated types that are composed of basic types. To improve efficiency, `TokenAware` introduces four optimizations. We conduct extensive experiments to evaluate `TokenAware` with real blockchain data. Results show that `TokenAware` can automatically identify new types of bookkeeping and recognize 107,202 tokens with 98.7% precision. `TokenAware` with optimizations merely incurs 4% overhead, which is 1/345 of the overhead led by the counterpart with no optimization. Moreover, we develop an application based on `TokenAware` to demonstrate how it facilitates malicious behavior detection.

CCS Concepts: • **Security and privacy → Software security engineering;**

Additional Key Words and Phrases: Ethereum, smart contract, token, bookkeeping recognition

This work is partially supported by Hong Kong ITF Project (Grant No. GHP/052/19SZ), the Research and Development Program of Shenzhen (Grant No. SGDX20190918101201696), Hong Kong RGC Projects (Grants No. PolyU15219319 and No. PolyU15224121), National Natural Science Foundation of China (Grants No. 61872057 and No. U19A2066), National Key R&D Program of China (Grant No. 2018YFB0804100), Natural Science Foundation of Sichuan Province (Grant No. 2022NSFSC0871).

Authors' addresses: Z. He, S. Song, W. Zhang, P. He, T. Chen (corresponding author), H. Li, and X. Zhang, University of Electronic Science and Technology of China, Chengdu, China; email: ecjgvmhc@gmail.com, {shuwei, 201821080421, 201822080439}@std.uestc.edu.cn, {brokendragon, hongweili, johnsonzxs}@uestc.edu.cn; Y. Bai, University of Electronic Science and Technology of China & Chengdu University of Information Technology, Chengdu, China; email: alicepub@163.com; X. Luo (corresponding author), The Hong Kong Polytechnic University, Hong Kong, China; email: csxluo@comp.polyu.edu.hk; X. Lin, University of Guelph, Guelph, Canada; email:xlin08@uoguelph.ca.

1049-331X/2023/02-ART26 $15.00
https://doi.org/10.1145/3560263

ACM Reference format:
Zheyuan He, Shuwei Song, Yang Bai, Xiapu Luo, Ting Chen, Wensheng Zhang, Peng He, Hongwei Li, Xiaodong Lin, and Xiaosong Zhang. 2023. TokenAware: Accurate and Efficient Bookkeeping Recognition for Token Smart Contracts. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 26 (February 2023), 35 pages.
https://doi.org/10.1145/3560263

1 INTRODUCTION

Token, a kind of cryptocurrency implemented by smart contracts (a.k.a., token smart contracts), lies at the heart of numerous blockchain-based applications, such as digital wallets, exchange markets, and blockchain explorers. Since tokens can be transferred from one account to another through the invocation of token smart contracts, recent years have witnessed the trend of tokenizing real-world assets (e.g., houses), so that they can also be traded via token exchanges [33]. Since Ethereum is the blockchain system hosting the most number of tokens [24], this work focuses on Ethereum tokens. The terms blockchain, smart contract refer to the Ethereum blockchain and the Ethereum smart contract hereinafter. Identifying token transfer behaviors, which usually involve money, value, and asset transfers, is essential to many applications, such as trading tokens in exchange markets, purchasing goods by tokens, predicting token prices [3, 10], facilitating the detection of various malicious behaviors relevant to tokens like money laundering [4], Ponzi schemes [2], token stolen [11], pump and dump [32], and so on.

Third-party tools can recognize token transfer behavior by monitoring standard interfaces and/or standard events (detailed in Section 9); however, many tokens do not strictly follow token standards, and the standard interfaces and standard events of these tokens do not necessarily reflect real token transfer behaviors [8], resulting in the inaccuracy of recognizing token transfer behaviors by monitoring standard interfaces and/or standard events. The experimental results in Section 6.1 also show that this approach has low accuracy (35.6%). The solution adopted by `TokenAware` overcomes this drawback: Since token smart contracts use internal bookkeeping to record the information of token holders (e.g., their addresses and shares) [8, 28], recognizing the bookkeeping and the operations on it empowers third-party tools to characterize the token transfer behaviors. Note that bookkeeping refers to the variables used to record the amount of tokens of each token holder, and bookkeeping recognition refers to identifying such variables. Token transfer behavior refers to the transfer of tokens, which is represented as a set of tuples $<addr, \delta>$. The tuple records the address of the token holder and the change of its token share. Moreover, a token transfer behavior necessarily modifies the bookkeeping as the shares of the recipient and sender need to be updated when the tokens are transferred.

Unfortunately, accomplishing this task is challenging, because smart contract bytecode does not contain type information. None of existing solutions can achieve it in an accurate and efficient manner because of their incomplete patterns and inefficient designs [8, 28]. For example, Frowis et al. propose a heuristic rule and apply it to **Ethereum Virtual Machine (EVM)** bytecode by symbolic execution and taint analysis [28]. However, their method has high false negative rate due to the imprecise and incomplete rule and the inherent limitations of symbolic execution and taint analysis [28]. The state-of-the-art tool, `TokenScope` [8], relies on four manually constructed patterns to recognize the bookkeeping. However, they are incomplete and manual bookkeeping recognition is time-consuming and error-prone. Moreover, since the data structures of bookkeeping variables may be changed with the evolution of smart contracts, manual pattern extraction cannot catch up with it. Besides, `TokenScope` was designed for offline analysis without any optimizations for improving the efficiency of bookkeeping recognition. Note that `TokenScope` is offline, because it first downloads and stores the execution traces of all transactions into a database and then analyses

these traces to detect the token transfer behavior, rather than detecting it while the smart contract is executing [8].

In this article, we propose and develop TokenAware, a novel online system for accurately and efficiently recognizing the bookkeeping and the operations on it for token smart contracts. By exploiting the observation that the data structure of bookkeeping can be either a few basic types (i.e., mapping, which acts as a hash table consisting of key types and corresponding value type pairs) or their combinations, TokenAware first learns the instruction sequences for locating basic types and then derives the instruction sequences for locating sophisticated types that are composed of basic types (in Section 4). After recognizing the bookkeeping, as a demonstration of characterizing token transfer behaviors, TokenAware captures token transfer behaviors by monitoring the operations on the bookkeeping, because a token transfer must modify the bookkeeping, which records the token amount of each token holder (Section 3.2). Moreover, by leveraging the unique features of blockchain and the modern multi-core architecture, TokenAware adopts four optimization mechanisms, including cache, parallelization, early extraction, and pipeline (in Section 5), for improving its performance. We conduct extensive experiments to evaluate TokenAware with real blockchain data consisting of millions of blocks. The experimental results show that TokenAware can automatically identify new types of bookkeeping and identify 107,202 tokens with 98.7% precision, which is 2.8× better than existing techniques. Moreover, TokenAware with full optimizations merely incurs about 4% overhead, which is 1/345 of the overhead led by the counterpart with no optimization. We also evaluate the effectiveness of each optimization mechanism and obtain several insights. To demonstrate how TokenAware can be used to detect malicious behaviors relevant to tokens, we develop an application based on TokenAware and use it to detect a new type of malicious behavior. It found 25 transactions used to launch the attacks, one of which has been reported [38]. These 25 transactions were sent to 17 smart contracts. By manually analyzing these transactions and smart contracts, we confirm that no false positives are produced. The source code and experimental results are available at https://github.com/hzysvilla/TokenAware.

In summary, this work makes three major contributions.

- We propose a novel approach to recognize the bookkeeping from the bytecode of a token smart contract.
- We develop TokenAware, a new automated system equipped with four optimization mechanisms to accurately and efficiently recognize the bookkeeping and operations on it (i.e., token transfer behaviors).
- We conduct extensive experiments to evaluate TokenAware using real blockchain data. The results show its accuracy, efficiency, and the effectiveness of each optimization mechanism.

2 BACKGROUND AND MOTIVATING EXAMPLE

2.1 Background Knowledge

The blockchain system is a decentralized ledger consisting of linked blocks, each of which contains zero or many transactions. The Ethereum blockchain is built on top of a P2P network consisting of many nodes [8]. When joining a blockchain system, a node synchronizes with other nodes by first downloading the blocks from other nodes and then executing the transactions in each block [8]. A smart contract is a program running on top of the blockchain, which is typically developed in high-level languages (e.g., Solidity) and then compiled into EVM bytecode for deployment.

Besides smart contract, Ethereum has another kind of account: **External-Owned Account (EOA)**, which does not contain executable code [19]. Each account is associated with a unique address. A transaction is a message for transferring cryptocurrencies, deploying a smart contract

```
1 mapping (address => uint256) private _balances;
2 address owner = 0x1122..;
3 function transfer(address _to, uint256 _value) public
    returns (bool success) {
4     emit Transfer(msg.sender, _to, _value);
5     _balances[msg.sender] = _balances[msg.sender].sub(_value);
6     _balances[owner] = _balances[owner].add(_value);
7 }
```

Fig. 1. An example token.

or invoking a smart contract. The transactions in Ethereum can be divided into external transactions, which are sent by EOAs, and internal transactions, which are sent by smart contracts. Only external transactions are recorded in the blocks whereas internal transactions can be recovered by re-executing smart contracts. To invoke a smart contract, an account should send a transaction that specifies the address of the invoked smart contract, the invoked function, and arguments.

Tokens are a kind of cryptocurrency implemented by smart contracts, which use internal bookkeeping to record the information of token holders (e.g., their addresses and shares) [8, 28]. The bookkeeping is recorded in the storage, a persistent and database-like space [51], because the modifications to the bookkeeping should be kept after the execution of token smart contracts. A token smart contract usually implements standard interfaces and emits standard events defined by token standards; otherwise, other blockchain-based applications cannot interact with the token. For example, if a token does not implement any token standard, it cannot be purchased or sold in exchange markets. The majority of tokens implement the ERC20 standard, which defines two standard interfaces transfer() and transferFrom() for transferring tokens, and one standard event Transfer for announcing the happened token transfer behaviors [50]. A token smart contract can also implement custom interfaces and events. Tokens can be transferred by invoking either standard interfaces or custom interfaces according to the implementations of token smart contracts.

Unfortunately, a recent study reveals that many tokens do not strictly follow token standard [8], and thus it is inaccurate to recognize token transfer behaviors by just monitoring standard interfaces and standard events. Instead, a better solution is to monitor the operations on the bookkeeping. We define the *token transfer behavior* incurred by one transaction as a set of tuples $<addr, \delta>$, recording the addresses of token holders and the changes of their token shares. If $\delta < 0$, then the corresponding account sends tokens; otherwise, the account receives tokens.

2.2 Motivating Example

Figure 1 shows an example implementation of the standard interface transfer(). Line 1 defines *_balances*, a bookkeeping variable whose type is mapping. *owner* defined in Line 2 is the address of an account. *_balances*[*owner*] (Line 6) is an item in the bookkeeping variable for recording the token amount possessed by *owner*. Lines 3 and 4 are the standard interface and the standard event, respectively, which indicate *_value* tokens are transferred from one account *msg.sender* to another account *_to*. To transfer tokens, *two* items in the bookkeeping variable are modified: one corresponding to the token sender (Line 5) and the other corresponding to the token receiver (Line 6). In this example, the account *owner* receives tokens rather than the account *_to* (Line 6), meaning that this example token does not strictly follow the standard.

After learning the instructions for locating a mapping item, `TokenAware` identifies that the token smart contract uses a mapping *_balances* as the bookkeeping (Section 3.1). When the standard interface transfer() is invoked, `TokenAware` notices that *_balances* is modified twice: one corresponding to Line 5 and the other corresponding to Line 6. From the first modification, `TokenAware` knows that

```
1 function doAirdrop(address[] memory srcs, address[] memory dests,
2 uint256[] memory values) onlyowner public nonPayable {
3     require(srcs.length > 0, 'Address arrays must not be empty');
4     require(srcs.length == dests.length, 'Address arrays must be of equal length');
5     uint256 i = 0;
6     while (i < srcs.length) {
7         emit Transfer(srcs[i], dests[i], values);
8         i++;
9     }
10    return i;
11 }
```

Fig. 2. The source code of the blockwell.ai KYC Casper Token. There exists no any token transfer behavior and the loop (lines 6 to 9) only send the standard event (line 7).

the account *msg.sender* is involved in the token transfer, because *msg.sender* is read by the instructions for locating a mapping item. Then, by applying use-definition chain analysis [34], TokenAware finds a SSTORE instruction, which takes in the result of such instructions. Please note that the SSTORE instruction is the only instruction for modifying the storage [51]. Therefore, TokenAware learns that the SSTORE instruction is used for modifying a bookkeeping item. After comparing the new value and original value stored in the bookkeeping item, TokenAware knows that the token amount of *msg.sender* decreases by *_value* and the token amount of *owner* increases by *_value*. Hence, the token transfer behavior by executing transfer() can be correctly recognized by TokenAware, which is {<*msg.sender*, −*_value*>, <*owner*, *_value*>}.

3 TOKENAWARE

TokenAware has two design goals. The first design goal is accuracy. That is, TokenAware must recognize the bookkeeping and the operations on it for token transfer accurately. Accurate recognition of token transfer behavior is extremely important, as malicious or irregular token transfer behavior can lead to user confusion as well as financial loss. For instance, the token smart contract named blockwell.ai KYC Casper Token cheated users by sending a standard event to inform others that tokens had been transferred, without actually transferring tokens [52]. More precisely, since this contract is closed source, with the help of decompilers [6, 30], we manually reverse engineer and comprehend the bytecode of blockwell.ai KYC Casper Token. The recovered code snippet of blockwell.ai KYC Casper Token is shown in Figure 2. We can see that only the standard events are emitted on line 7 without token transfer operations in the loop from line 6 to line 9. If someone invokes the method doAirdrop() (line 1), then the victim's wallet will observe the standard event and inform mistakenly that the corresponding token transfer has happened. Such fraudulent implementation can lead to user confusion and financial loss. The second design goal, namely, efficiency, is presented in Section 3.3.

Figure 3 depicts the architecture of TokenAware, which is implemented in an Ethereum node.

TokenAware fetches the data from Ethereum node for analyzing the token transfer behaviors on Figure 3. TokenAware first monitors external transactions and internal transactions to extract the EVM bytecode of smart contracts. Specifically, because the analysis target of TokenAware is EVM bytecode of token contracts, TokenAware will monitor all the operations of external and internal transactions. If these transactions are monitored to create contracts, then TokenAware will extract the EVM bytecode of these newly created contracts. Next, on phase 1 of Figure 3, TokenAware appends the extracted bytecode to the task queue for bookkeeping recognition (detailed in

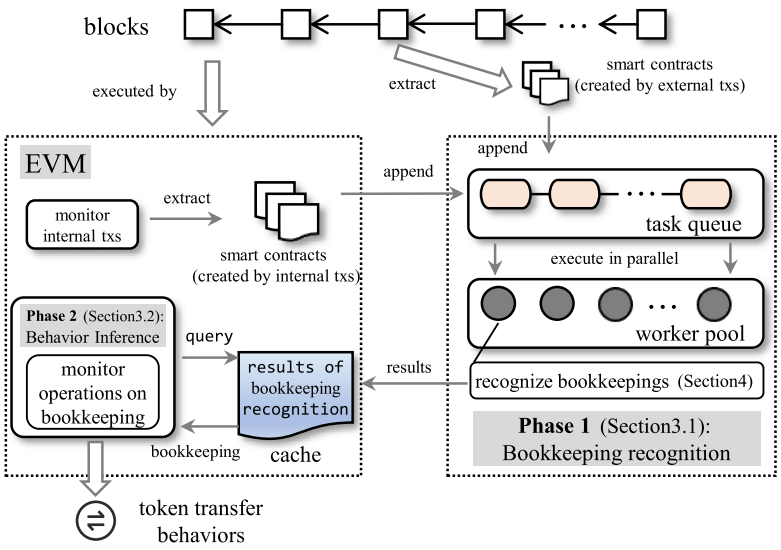


Fig. 3. Architecture of TokenAware.

Section 3.1). Specifically, TokenAware prepares a worker pool consisting of multiple workers, each of which binds to a thread. Besides, TokenAware considers bookkeeping recognition of each bytecode as a task, and uses one worker to execute a task. After that, TokenAware recognizes the bookkeeping in four steps (detailed in Section 4), and this phase acquires the results of bookkeeping recognition and records them in the cache as input of the next phase. As for the phase 2 on Figure 3, TokenAware will monitor the execution of token contracts. Specifically, when transactions invoke these token contracts, TokenAware will analyze whether these transactions modify the bookkeeping. TokenAware analyzes modification operations to bookkeeping through the relational pattern (detailed in Section 4) of EVM instruction sequences. Finally, TokenAware will infer the token transfer behavior from the modifications of the bookkeeping (detailed in Section 3.2).

This section introduces the design of TokenAware, and leaves the recognition of bookkeeping and four optimizations in next sections. Currently, TokenAware supports the tokens implementing the ERC20 standard, which is the most popular token standard. How to extend TokenAware to support other token standards is discussed in Section 4.6.

3.1 Phase 1: Bookkeeping Recognition

After downloading blocks from other nodes, TokenAware applies an optimization technique (i.e., **O3**: early recognition) to recognize bookkeeping variables before the transactions in the blocks are executed by EVM. Particularly, TokenAware regards the bookkeeping recognition of each smart contract as a task, and appends each task to a task queue. TokenAware applies another optimization technique (i.e., **O2**: parallelization) by preparing a worker pool consisting of multiple workers and executing tasks by workers in parallel. A worker is responsible for processing one smart contract.

The results of bookkeeping recognition containing two kinds of information are recorded in a cache. The first is a pair associating the hash of a smart contract's bytecode with a bool value indicating whether the smart contract is a token. If the bool value is true, then the second information, which is the recognized bookkeeping as well as the instructions sequence to locate an item of the bookkeeping, exists. TokenAware queries the cache to check whether the bytecode of a smart contract has already been analyzed. If so, then the bytecode will not be analyzed again.

```
1 func opAdd(......) (...) {
2     x, y := stack.pop(), stack.peek()
3     //monitor the add instrucion
4     TokenAware.Collect("ADD", common.BigToHash(x),common.BigToHash(y))
5     math.U256(y.Add(x, y))
6     interpreter.intPool.put(x)
7     return nil, nil
8 }
```

Fig. 4. TokenAware monitors the implementation of EVM instructions, taking the instruction of ADD as an example. The function opAdd() is the implementation of the EVM instruction ADD, and we insert the code stub (i.e., Collect() method) of the TokenAware on highlighting line 4.

Internal transactions are not recorded in the blocks, and they can also be used for deploying smart contracts [51]. Therefore, the smart contracts created by internal transactions cannot be processed by early recognition. Instead, TokenAware monitors the execution of all internal transactions, and extracts the EVM bytecode from each internal transaction for deploying a smart contract. The extracted bytecode is also appended to the task queue for bookkeeping recognition.

Automated bookkeeping recognition is challenging, because EVM bytecode does not contain type information. TokenAware overcomes the challenge by first learning the instruction sequences for locating basic types and then deriving the instruction sequences for locating sophisticated types that are composed of basic types. We elaborate more on the four steps of bookkeeping recognition in Section 4.

3.2 Phase 2: Behavior Inference

We insert code stubs to the Ethereum node, which can obtain the execution chance during the execution of smart contracts, so TokenAware can monitor and pause the execution of smart contracts. Whenever a smart contract A is invoked, TokenAware will pause the execution of A by blocking the EVM process, compute the hash of A's bytecode, and then query the result from the cache. Specifically, in the design of the EVM, there are four kinds of instructions that can invoke contracts, including CALL, DELATEGATECALL, CALLCODE, and STATICCALL. Among them, STATICCALL cannot access the bookkeeping, because it is an instruction for contract to call other contract without modifying state. So, we insert the code stub of TokenAware in the implementation of the remaining three instructions that can invoke the contract. Taking the instruction of CALL as an example, whenever the CALL instruction invokes a contract, it will first obtain the bytecode of the contract. After obtaining the bytecode, TokenAware will execute the code of the caching mechanism, including calculating the hash of the bytecode, querying whether the hash hits, whether to wait, and so on.

TokenAware applies the optimization technique (i.e., **O4**: pipeline), which enables us to recognize the bookkeeping from a smart contract and in the meanwhile run the subsequent transactions, which do not invoke the contract. Hence, pipeline speeds up TokenAware, because the subsequent transactions may not need to wait for the result of bookkeeping recognition.

There are three kinds of query results. The first is the bookkeeping of A, which means that A is an ERC20 token and its bookkeeping has been recognized. In this case, TokenAware monitors the execution of A and infers the token transfer behavior from the modifications to the bookkeeping. By adding code to the EVM instructions to record the execution of each instruction, we first monitor the execution of the instruction sequence (termed by *INS*) for locating an item of the bookkeeping.

Taking the instruction of ADD as an example, we display the monitor implementation in Figure 4. The function opAdd() on Figure 4 is the EVM implementation of the instruction

```
1 func (*TokenAware) Collect(......) (...) {
2     if _,Existflag:=containerOpcodes[OpCode];Existflag{
3        containerflags:=containersAnalysis(priorSeq,Opcode,traces)
4        if len(ret)>0{
5           UpdateState(PriorSeq,Opcode,traces,containerflags)
6        }
7     }
8 }
```

Fig. 5. The code snippet of the Collect() method in TokenAware.

ADD, and we insert the monitor code snippets (i.e., Collect() method) on line 4. More precisely, whenever the contract executes the ADD instruction, TokenAware can monitor the instructions to collect the opcode information and operands by using the method Collect() (line 4).

Figure 5 demonstrates the code snippet of the method Collect(), which is mainly used to monitor and pre-process the information about the EVM instructions. Specifically, it will identify whether the current opcode (i.e., the Opcode variable) is one of the instruction sequence (the priorSeq variable) for locating an item of the bookkeeping as shown in line 3 of Figure 5. We explain the identification rule in Section 4. Once TokenAware recognizes that the current opcode is used to locate an item of the bookkeeping, it will append the opcode (i.e., the Opcode variable) with their operands (i.e., traces variable) to the instruction sequence (i.e., the priorSeq variable) on the line 5. When the next opcode comes, such implementation (i.e., UpdataState()) ensures the instruction sequence (i.e., the priorSeq variable) has been updated.

Then, we know the account (termed by *addr*) involved in the token transfer behavior, because *INS* reads the address of a token holder. Besides, we apply use-definition chain analysis [34] to examine which SSTORE instruction uses the computation result of *INS*. Since *INS* locates an item of bookkeeping, the SSTORE instruction modifies the item. If such SSTORE is found, then we first read the value recorded in the location to be written, which indicates the token amount of the token holder before the token transfer behavior happens. Then, we obtain its second operand (its first operand is the location to be written), which is the value to be written, indicating the new token amount of the token holder after the token transfer happens. With the new amount and the original amount, we obtain the change (termed by δ) of tokens possessed by the token holder. Since the token transfer behavior can result in the modifications to the token amounts of multiple accounts, TokenAware will monitor multiple operations to the bookkeeping, which is a set of tuple $<addr, \delta>$.

The second is a bool value *false*, indicating that A is not an ERC20 token. In this case, TokenAware lets A run as usual without monitoring. The third is empty, which means that TokenAware has not finished bookkeeping recognition from A. In this case, TokenAware waits for a short while, 1 ms in the current implementation, and then queries the cache again.

3.3 Basic Idea of Optimization

The second design goal of TokenAware is efficiency. That is, TokenAware should not incur high overhead to an Ethereum node to support online scenarios. Improving the efficiency of recognizing token behavior is crucial, as many applications require real-time visibility into token transfers. For example, a store application should deliver the items to buyers once the receipt of the tokens is confirmed. Furthermore, various malicious behaviors relevant to tokens need to be detected quickly so that token holders or stakeholders can take timely countermeasures. More specifically, it is commonly accepted that a transaction is confirmed or irreversible when the number of blocks

mined after the block containing this transaction exceeds 7 [21]. It is around 91 s, since it takes about 13 s to mine a new block [20]. Our experiments show that the overhead of TokenScope is 12.9 (Section 6.2), so it takes about 167.7 (12.9×13) s to detect the token transfer behavior in a transaction. That is, when TokenScope identifies a token transfer, the transaction is already confirmed and irreversible. Imagine that malicious behavior is involved in this transaction, then there is no time to take countermeasures to avoid financial losses before the transaction is confirmed. To address the above problem, TokenAware should be designed as an online method that does not incur high overhead.

Therefore, we propose four optimization mechanisms to improve the efficiency of TokenAware by fully exploiting unique features of Ethereum and the modern multi-core architecture. First, as code clone is prevalent in smart contracts [31, 37], TokenAware only processes smart contracts with unique bytecode. Second, given that Ethereum node does not fully use the multi-CPU and multi-core architecture of modern computers, TokenAware makes full use of the spare computing resources by parallelization for efficiency. Third, since the bytecode of smart contracts can be obtained before executing the external transaction, TokenAware can perform bookkeeping recognition when transactions are waiting for execution. Fourth, TokenAware supports recognizing the bookkeeping from a smart contract and in the meanwhile running the subsequent transactions, which do not invoke the contract. A more detailed description of the above four optimization mechanisms is given in Section 5.

4 BOOKKEEPING RECOGNITION

TokenAware recognizes the bookkeeping from EVM bytecode rather than source code, because open-source smart contracts only account for less than 1% of total smart contracts [35]. Automated bookkeeping recognition is challenging, because EVM bytecode does not contain type information. We overcome the challenge based on four observations. First, the bookkeeping should be a basic container or the combination of basic containers (detailed in Section 4.1); otherwise, the bookkeeping cannot record the token information of multiple token holders. Second, since there are only four basic containers in Solidity, moderate effort is needed to learn from basic containers. Third, we can differentiate the types of different basic containers if the instruction sequences for locating an item of them are different. Fourth, the types of basic containers and the instruction sequences for locating an item of them are not changed with the evolution of Solidity. Therefore, our approach is applicable to all existing Ethereum smart contracts.

Bookkeeping recognition consists of four steps. First, we learn the instruction sequence for locating an item of each basic container (Section 4.2). Second, we derive the instruction sequences for locating an item of all possible combinations of basic containers by combing the instructions for locating an item of basic containers (Section 4.3). Third, we search the EVM bytecode for all derived instruction sequences to determine which instruction sequences are used in the smart contract and how the instruction sequence is combined (Section 4.4). Finally, we recognize the bookkeeping of the smart contract by checking how the instruction sequence is combined (Section 4.5). Furthermore, we illustrate how TokenAware recognizes bookkeeping with a real deployed token smart contract (Section 4.7).

4.1 Two Requirements of the Bookkeeping

Bookkeeping refers to variables used in token smart contracts to record the number of tokens of each token holder, and it should satisfy two requirements. R1: the bookkeeping should be a container, and the term "container" is commonly used in programming to refer to a data structure whose instances are collections of other objects [27, 42]. Specifically, the bookkeeping holds multiple objects, each of which records the quantity of tokens owned by a holder and is indexed by the

holder's identifier (i.e., address). Such a design allows a token smart contract to access a particular holder's balance by retrieving the objects in the bookkeeping. R2: the size of the bookkeeping should be dynamic. Since token smart contracts are developed with no certainty of how many holders there will be in the future, a fixed-size data structure cannot be used as the bookkeeping. Otherwise, the balance of new holders cannot be recorded when the number of holders reaches the capacity of that data structure.

By carefully and completely consulting Solidity's official documentation [22], we find four basic containers, namely, struct, one-dimensional static array, mapping, and one-dimensional dynamic array (Section 4.2). For example, a variable of mapping type can store a collection of key-value pairs. It is worth noting that since the combination of these four basic containers is a collection of other objects (i.e., basic containers), the combination is still a container. Please recall R1, i.e., the bookkeeping should be a container. Hence, bookkeeping should be either a basic container or a combination of them.

Please note that these two requirements are for bookkeeping rather than the basic container. For example, the two basic containers, struct and one-dimensional static array, have fixed sizes. As the size of bookkeeping should be dynamic to support adding and removing accounts, struct and one-dimensional static array alone cannot constitute bookkeeping; they must be combined with two other basic containers with dynamic sizes (i.e., mapping and one-dimensional dynamic arrays) to construct bookkeeping.

4.2 Basic Containers

A *basic* container does not contain other containers. We find that there are only four basic containers in Solidity. To learn the instruction sequence (with at least one instruction) for locating an item of each basic container, we first write Solidity code, which defines and initializes the basic container and modifies one of its items, then compiles the source code into EVM bytecode, and finally obtain the memory layout and the instruction sequence by reverse engineering the bytecode.

Two basic containers, struct and one-dimensional static array with fixed sizes, are usually used to record the token amount of one account. The other two, mapping and one-dimensional dynamic array with extensible sizes at runtime, are commonly used to record multiple items, each of which may contain the token amount of one account. We do not regard a multi-dimensional array as a basic container, because it can be regarded as the combination of multiple one-dimensional arrays. We find that the instruction sequence for locating an item of a one-dimensional static array is the *same* as that of a struct, because a one-dimensional static array can be regarded as a struct whose items are of the same type. For instance, Figures 6(a) and 6(b) show a struct (i.e., *StructSample*) containing three unsigned integer items and a one-dimensional static array (i.e., *ArraySample*) that also contains three elements of the same type. As shown in Figure 6(c), the layout of this struct in the memory has three fields, i.e., three unsigned integers (i.e., *a*, *b*, and *c*), which are arranged in order. The layout of the array in the memory is the same as that of the struct, as shown in Figure 6(d). The instruction sequence used for accessing a variable is determined by the layout of the variable, so the instruction sequence to locate an item of a struct is the same with that of a one-dimensional static array. To learn the instruction sequences for locating items of such two containers with the same layout, we first write Solidity code, which defines and initializes the above struct and array (i.e., *StructSample* and *ArraySample*) and modifies one of their items, then compiles the source code into EVM bytecode. Finally, by reverse engineering the bytecode, we find that the instruction sequence for locating an item of *StructSample* is the same as that of *ArraySample*.

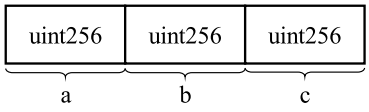
—*Struct.* A struct variable has a unique id, and the location of a struct item is the summation of the id and the offset from the item to the beginning of the struct variable. Figure 7(b) shows the

```
1  struct StructSample {
2      uint256 a;
3      uint256 b;
4      uint256 c;
5  }
```

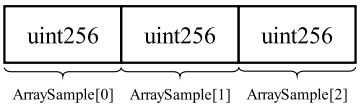
(a) A struct with three items

```
1  uint256[3] ArraySample;
```

(b) An array with three elements

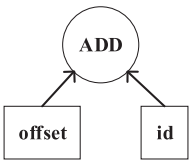


(c) The layout of *StructSample*



(d) The layout of *ArraySample*

Fig. 6. The layout of a struct containing three unsigned integers is the same as that of a one-dimensional static array containing three unsigned integers.



(a) Tree format

Nr.	Instruction
01	PUSH offset
02	PUSH id
03	ADD

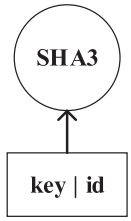
(b) Disassemble text

Fig. 7. Instructions to locate an item of a struct.

instruction sequence for locating an item of a struct, where ADD is the instruction for addition [51]. The number before each instruction specifies the execution order. We organize the instruction sequence as a tree (Figure 7(a)) to ease the combination of the instruction sequences for locating an item of multiple containers (in Section 4.3). Each circle in the tree denotes an EVM instruction, and each rectangle represents data. An edge from the data to an instruction indicates that the data is an operand of the instruction. If an instruction takes in more than one operand, then these operands are displayed in order of appearance from left to right. It is worth noting that we organized the instruction sequence into a tree by only retaining the instructions highlighted in gray. We ignore the following instructions, PUSH for pushing a number on the stack, DUPx $1 \leq x \leq 16$ for duplicating one stack item, SWAPx $1 \leq x \leq 16$ for exchanging two stack items, AND for bitwise AND operation, MSTORE for saving word to memory, LT for less-than comparison, because these instructions are frequently used in all smart contracts and they cannot distinguish locating an item of the bookkeeping from other operations.

—*Mapping*. A mapping variable has a unique id, and each item maps a key to a value. Figure 8 shows the instruction sequence for locating the value of a mapping item. An instruction SHA3 is used to compute a hash, which is the location of the value. The operand of the SHA3 consists of two parts. The first part is the key, and the second part is the id of the mapping variable.

—*One-dimensional dynamic array*. Figure 9 illustrates the instruction sequence for locating an item of a one-dimensional dynamic array. An edge from a circle (i.e., the instruction) to a rectangle (i.e., the data) indicates that the data is the result of the instruction. Every one-dimensional dynamic array has a unique id. The length of the array is recorded in the location, which is the result of a SHA3 whose operand is the id. Array items lay immediately after the location recording the array length. Hence, the location of an array item is the summation of the SHA3 result with the offset from the array item to the location of the array length.

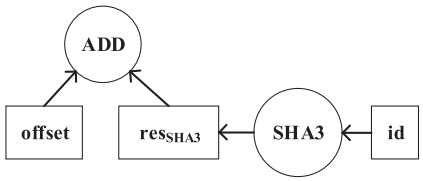


(a) Tree format

Nr.	Instruction	Nr.	Instruction	Nr.	Instruction
01	PUSH id	07	AND	13	DUP2
02	PUSH 0	08	DUP2	14	MSTORE
03	DUP4	09	MSTORE	15	PUSH 20
04	PUSH 0xFF…	10	PUSH 20	16	ADD
05	AND	11	ADD	17	PUSH 0
06	PUSH 0xFF…	12	SWAP1	18	SHA3

(b) Disassemble text

Fig. 8. Instructions to locate an item of a mapping.



(a) Tree format

Nr.	Instruction	Nr.	Instruction	Nr.	Instruction
01	PUSH id	07	SWAP1	13	ADD
02	DUP3	08	PUSH 0		
03	DUP2	09	MSTORE		
04	SLOAD	10	PUSH 20		
05	DUP2	11	PUSH 0		
06	LT	12	SHA3		

(b) Disassemble text

Fig. 9. Instructions to locate an item of a one-dimensional dynamic array.

4.3 Container Combinations

Since the bookkeeping can also be the combination of basic containers, we derive the instruction sequences for locating an item of container combinations by merging the trees that represent the instruction sequences for locating an item of basic containers. Specifically, we identify *two* modes of container combinations (**C1** and **C2**) and design different merging strategies for them.

—**C1** (variable composition mode). In such a combination, each basic container is a variable, and one container variable reads the value of the other container variable. The instruction sequence for locating an item of a variable composition is derived by three steps. First, a rectangle representing the value of one container variable and an edge connecting the tree of the container variable to the rectangle are added. Second, a SLOAD instruction and an edge connecting the rectangle added in the first step to SLOAD circle are added, indicating that a value in a container variable is read [51]. Third, an edge connecting the SLOAD circle to the tree of a container is added, indicating that the value read by the SLOAD instruction is used by the connected pattern.

Figure 17(a) shows the source code of a token smart contract whose bookkeeping is the composition of two variables, which are two basic containers, *userID* (Line 1) and *balances* (Line 2). To transfer tokens, the value *id* in the container *userID* is read (Line 4), and *id* is used for accessing the other container *balances* (Line 5). Figure 10 depicts the instruction sequence for locating an item of the two bookkeeping variables, which is merged from the instruction sequence for locating an item of *userID* (i.e., the subtree within the left circle) and the instruction sequence for locating an item of *balances* (i.e., the subtree within the right circle). The *key* is the address of a token holder. In the first step, we add a rectangle res_{SHA3} and connect the left subtree to the rectangle, indicating that res_{SHA3} is the computation result of SHA3. In the second step, we add a circle SLOAD and connect res_{SHA3} to it, which means that SLOAD reads a value whose location is res_{SHA3}. Finally, we connect SLOAD to the right subtree, indicating that the read value is used by *balances*.

—**C2** (nested container mode). In such a combination, both combined containers belong to the same variable. The instructions for locating an item of a nested container is directly merged from the instructions for locating an item of the combined containers. Specifically, we connect a

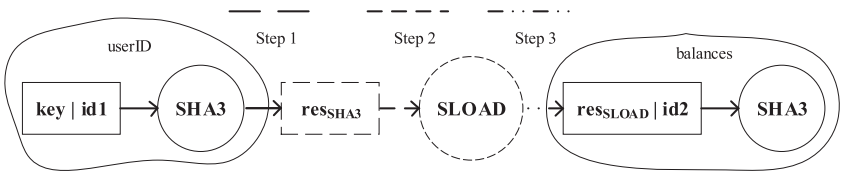


Fig. 10. Instructions to locate an item of a container combination (**C1**).

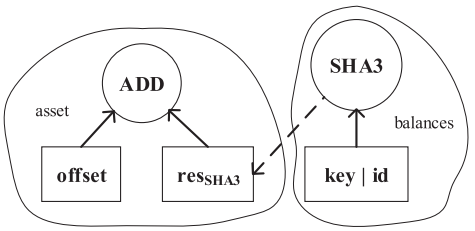


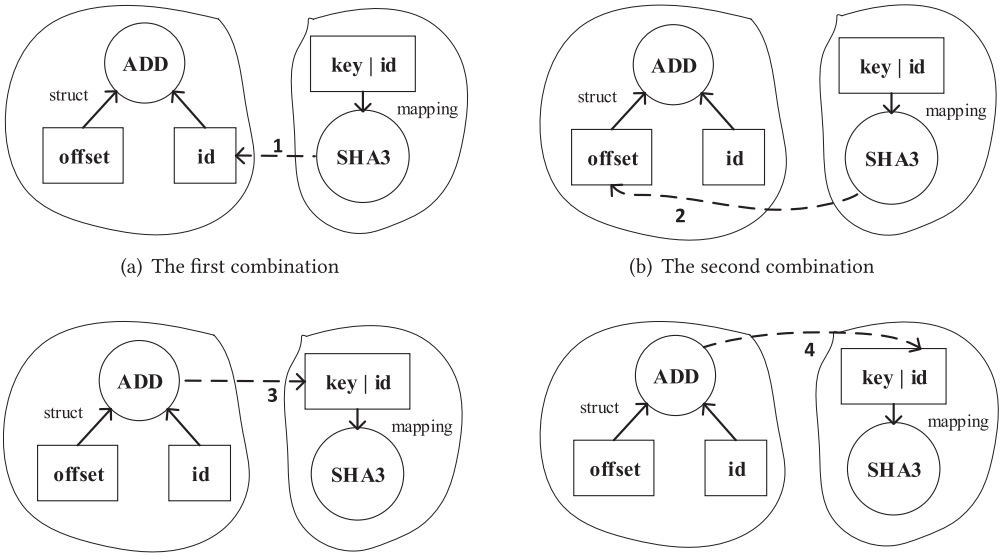
Fig. 11. Instructions to locate an item of a container combination (**C2**).

rectangle of one tree to a circle of another tree, indicating that the data in the rectangle is computed by the instruction in the circle. Figure 17(b) shows the source code of a token smart contract whose bookkeeping is a nested container, a mapping associating an address to a struct asset. Figure 11 depicts the instructions for locating an item of the bookkeeping. The subtrees within the left circle and the right circle are the instructions for locating an item of the struct and the mapping, respectively. The *key* is the address of a token holder. The dashed edge means that the result of SHA3 is used by ADD.

A non-basic container can be considered as a container that is combined from a container with a basic container according to the two composition modes. Therefore, for any container combination *co*1 that is combined from a container *co*2 with a basic container *b*, we obtain the instruction sequence for locating an item of *co*1 by combing the instruction sequence for locating an item of *co*2 with the instruction sequence for locating an item of *b*. After iterative combinations, a complicated container can be produced.

We use n to represent the number of combination operations and f_n to represent the number of containers that are produced by n combination operations, as shown in Equation (1). Obviously, $f_0 = 4$, because there are four basic containers. We then explain how f_n can be derived from f_{n-1}. As shown in Figures 7(a), 8(a), and 9(a), each basic container has two leaf operands (i.e., the operands in the leaf nodes). We consider that the SHA3 instruction to access a mapping item (Figure 8) takes in two leaf operands, because its operand consists of two parts, *key* and *id*. Let us first consider the combination of two containers in the nested container mode. Suppose the two containers have x, y leaf operands, respectively, they have x+y container combinations, because they are combined by connecting each instruction in the root node to the leaf operands of the other container.

Figure 12 illustrates the container combinations of a mapping and a struct in the nested container mode. There are four (i.e., 2+2, both two basic containers have two leaf operands) combinations. The first combined container is the one shown in Figure 11. The second combined container means that the offset of the struct is the computation result of a SHA3 instruction (shown in Figure 12(b)). The remaining two combined containers (Figures 12(c) and 12(d)) can be explained similarly. Each container after n combination operations has $n + 2$ leaf operands, because each combination operation adds one leaf operand. Figure 11 shows a container after one combination, which has three (i.e., 1+2) operands, *offset*, *key*, and *id*.



(a) The first combination (b) The second combination

(c) The third combination (d) The fourth combination

Fig. 12. Container combinations of a mapping and a struct.

Therefore, the number of containers that are combined from a container after $n-1$ combinations and one basic container in the nested container mode is $n+3$, because the former has $n+1$ leaf nodes and the latter has two leaf nodes. Such number is also $n+3$ if these two containers are combined in the variable composition mode. We omit the derivation process, because it is similar to the derivation of the nested container mode. Please recall that a container after n combinations are combined from a container after $n-1$ combinations and a basic container, and hence f_n should be $8(n+3)f_{n-1}$, which is the product of f_{n-1} ((i.e., the number of containers after $n-1$ combinations), 4 (i.e., the number of basic containers), $n+3$ ((i.e., the number of containers that are combined from a container after $n-1$ combinations and one basic container in the nested container mode), and 2 (i.e., the two combination modes produce the same number of containers). After expanding Equation (1), we obtain the result that $f_n = \frac{2}{3} \times 8^n(n+3)!$.

$$\begin{cases} f_n = 8(n+3)f_{n-1}, n > 0, \\ f_0 = 4. \end{cases} \tag{1}$$

In our research on token smart contracts, we find that n is very small in practice. For example, $n = 2$ for all four kinds of bookkeeping introduced by `TokenScope` [8]. To the best of our knowledge, the value of n is no greater than three in currently deployed token smart contracts. Therefore, we set the upper bound of n as three by default. Actually, n is a tunable parameter for `TokenAware`, and we allow users to adjust it to cope with more complex token smart contracts that may arise in the future. Hence, for a given n, we obtain $\sum_{i=0}^{n} f_i$ instruction sequences, each of which is used for locating an item of a container.

4.4 Searching Bytecode for Instruction Sequences of BabyDTXToken

Our approach to search the bytecode of a smart contract for bookkeeping takes in the bytecode b, and the set of instruction sequences for locating an item of containers ap, derived from container combinations (Section 4.3). Our approach consists of three steps. First, it eliminates the smart contract without implementing two standard interfaces, transfer() and transferFrom(), and without

emitting the Transfer event, because an ERC20 token must implement at least one of them [50]. Second, we check whether b contains an instruction sequence p that belongs to ap. If so, then in the third step, we check whether p can be executed at least two times for any contract invocation. If so, then we consider p as the instruction sequence for locating an item of the bookkeeping in b, because a token transfer behavior should modify at least two bookkeeping items for the token sender and receiver, respectively.

Step 1. If a smart contract implements an interface or emits an event, then the bytecode of the contract must contain the hash of the interface's signature (i.e., the name of the interface and the list of parameter types) or the hash of the event's signature (i.e., the name of the event and the list of parameter types) [51]. Hence, this step simply searches b for the hashes of transfer(), transferFrom(), and Transfer. If none can be found, then the contract does not implement the ERC20 standard.

Step 2. Our approach interprets the EVM instructions of b, because the operands and the computation results of EVM instructions are needed for use-definition chain analysis [34]. During interpretation, our approach tries to search b for a p that belongs to ap. A straightforward approach is matching b with every p in ap, however, such approach is impractical due to the large number of all possible types of bookkeeping variables (described in Section 4.3). We apply an efficient method by first merging all instruction sequences in ap into three trees whose roots are the instruction sequences of three basic containers, because each instruction sequence is combined from the instruction sequences of simpler containers. Then, searching b for a p is equivalent to finding a path in the trees, whose complexity is $O(n)$ where n is the number of combination operations. Once we find p, we also know the code blocks containing p in b.

Step 3. A smart contract contains three kinds of functions, public, external, and internal [19]. Only public functions and external functions can be invoked by transactions [19]. Therefore, this step first identifies the code region that can be executed when a given public/external function is invoked. We achieve this goal by leveraging a decompiler [30]. After that, we count the execution number of the code blocks containing p (obtained in step 2) within the code region for each public/external function. If the number exceeds one, then we consider p as the instruction sequence to locate an item of the bookkeeping.

4.5 Recognize Bookkeeping from Instructions

After finding the instructions sequence p from the smart contract (Section 4.4), it is straightforward to recognize bookkeeping by parsing p, because (1) we know how p is combined from the instruction sequences to locate an item of basic containers (Section 4.3), and (2) we know the instruction sequence corresponding to each basic container (Section 4.2). When encountering an instruction sequence shown in Figure 7, we recognize a struct no matter whether the container is declared as a struct or a one-dimensional static array in the source code (`TokenAware` directly processes EVM bytecode), because the instruction sequences to locate an item of them are the same. The accuracy of `TokenAware` will not be affected by the indistinction of a struct and a one-dimensional static array, because `TokenAware` recognizes token transfers by monitoring the execution of EVM instructions.

4.6 Extension to Other Token Standards

Although this work focuses on ERC20 tokens, it is straightforward to extend our method to support other token standards. Please recall that the only step in our approach requiring the specification of the ERC20 standard is the step 1 in Section 4.4, which is used for eliminating non-token smart contracts and non-ERC20 tokens. Hence, to support another token standard A, we just need to consider the standard interfaces and standard events defined by A in step 1.

```
1  contract Costume {
2      mapping (uint256 => mapping(address => uint256)) internal balances;
3      event TransferSingle(address indexed _operator, address indexed
   _from, address indexed _to, uint256 _id, uint256 _value);
4      event TransferBatch(address indexed _operator, address indexed
   _from, address indexed _to, uint256[] _ids, uint256[] _values);
5      function safeTransferFrom(address _from, address _to, uint256 _id,
   uint256 _value, bytes calldata _data) external {
           ...
6          balances[_id][_from] = balances[_id][_from].sub(_value);
7          balances[_id][_to]   = _value.add(balances[_id][_to]);
8          emit TransferSingle(msg.sender, _from, _to, _id, _value);
           ...
9      }
10     function safeBatchTransferFrom(address _from, address _to, uint256[]
   calldata _ids, uint256[] calldata _values, bytes calldata _data)
   external {...}
11 }
```

Fig. 13. Code snippet of an ERC1155 token.

We use a real token smart contract to show that TokenAware can be easily extended to ERC1155 token standard, which is the standard for contracts that manage multiple types of tokens [46]. Figure 13 shows the (simplified) code snippet of an ERC1155 token named Costume. ERC1155 allows this single contract to manage multiple types of tokens, which are uniquely identified by their ids (i.e., *_id* in line 3) [46]. Costume's container consists of two mappings nested together (line 2). The outer mapping maps the token's id to the inner mapping, while the inner mapping maps the token holder to its shares. Thus, *balances[_id][_owner]* represents the number of tokens with identifier *_id* owned by the holder *_owner*. Function safeTransferFrom() (line 5) is called to transfer a certain type of token. The arguments *_from*, *_to*, *_id*, and *_value* indicate that *_value* tokens with the identifier *_id* are transferred from holder *_from* to holder *_to*. The function safeBatchTransferFrom() (line 10) works very similar to safeTransferFrom(), the only difference is that it allows transferring multiple types of tokens at once. The event TransferSingle (line 3) or TransferBatch (line 4) must be emitted when tokens are transferred.

Since the only step in our approach that requires the specification of a standard is step 1 in Section 4.4, we can enable TokenAware to identify whether a contract implements the ERC1155 standard by leveraging the two standard interfaces and two standard events mentioned above in this step. More precisely, this step simply searches the bytecode *b* for the hashes of safeTransferFrom(), safeBatchTransferFrom(), TransferSingle, and TransferBatch. If none can be found, then the contract does not implement the ERC1155 standard. The other parts of TokenAware do not need to be modified, because they do not involve the specification of a standard.

4.7 Example of Bookkeeping Recognition

In Figure 14, we introduce an ERC20 token contract called BabyDTXToken to show the workflow of TokenAware. The BabyDTXToken refers to the DTXToken,[1] which is actually deployed on Ethereum. We next introduce how TokenAware recognizes the bookkeeping in BabyDTXToke. First,

[1]The address of DTXToken on Ethereum is 0x765f0c16d1ddc279295c1a7c24b0883f62d33f75.

```
contract BabyDTXToken {

    struct  Checkpoint {
        uint fromBlock;
        uint value;
    }
    //bookkeep
    mapping (address => Checkpoint[]) balances;
    event Transfer(address indexed _from, address indexed _to, uint256 _amount)

    function transfer(address _to, uint256 _amount) public returns (bool success) {
          ……
          uint previousBalanceFrom = balanceOfAt(_from, block.number);
          if (previousBalanceFrom < _amount) {
              return false;
          }
          updateValueAtNow(balances[_from], previousBalanceFrom - _amount);
          uint previousBalanceTo = balanceOfAt(_to, block.number);
          require(previousBalanceTo + _amount >= previousBalanceTo);
          updateValueAtNow(balances[_to], previousBalanceTo + _amount);
          Transfer(_from, _to, _amount);
          return true;
    }

    //Queries the balance of `_owner` at a specific `_blockNumber`
    function balanceOfAt(address _owner, uint _blockNumber) public constant
        returns (uint) { …… }

    function updateValueAtNow(Checkpoint[] storage checkpoints, uint _value
    ) internal  {
        if ((checkpoints.length == 0)
        || (checkpoints[checkpoints.length -1].fromBlock < block.number)) {
              Checkpoint storage newCheckPoint =
checkpoints[ checkpoints.length++ ];
              newCheckPoint.fromBlock =  uint128(block.number);
              newCheckPoint.value = _value;
          } else {
              Checkpoint storage oldCheckPoint = checkpoints[checkpoints.length-1];
              oldCheckPoint.value = _value;
          }
    }
}
```

Fig. 14. The code segment of BabyDTXToken.

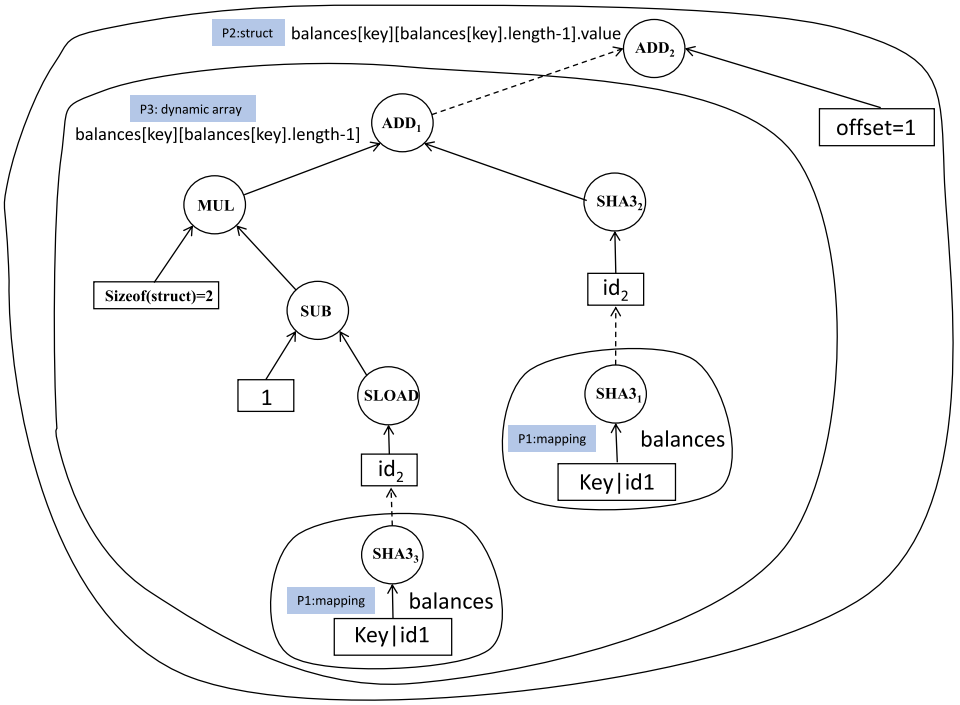


Fig. 15. The tree format bookkeeping of BabyDTXToken.

we explain the BabyDTXToken's container combination in Section 4.7.1, analyze the instruction sequence search at Section 4.7.2, and then show the process of recognizing bookkeeping from the instructions on BabyDTXToken at Section 4.7.3.

4.7.1 Container Combinations of BabyDTXToken. The bookkeeping of BabyDTXToken contract is defined in Figure 14, Line 8. In the bookkeeping definition (i.e., *mapping (address => Checkpoint[])*) of BabyDTXToken contract, we can find three basic containers including mapping type (i.e., *mapping*), struct type (i.e., *Checkpoint*), and one-dimensional dynamic arrays type (i.e., *Checkpoint[]*). The bookkeeping of BabyDTXToken maps token holders with a dynamic array of structures that records the historical balances of token holders. From Figure 14, we can see that the usage of BabyDTXToken's bookkeeping is complex, requiring the invocation of multiple functions (e.g., transfer(), updateValueAtNow(), balanceOfAt()). Therefore, we eliminate intermediate variables and simplify the use of bookkeeping for BabyDTXToken to the simple statement (i.e., *balances[key][balances[key].length-1].value*). The key is the address of a token holder.

Figure 15 depicts the instructions for locating an item of the bookkeeping on BabyDTXToken. The tree with the root as ADD_2 in Figure 15 corresponds to the struct container (depicted in Figure 7), its left subtree is the calculation of the position of the dynamic array container type (i.e., *id* in Figure 7) and variable of the right subtree in is offset in struct container. The tree with the root as ADD_1 in Figure 15 corresponds to the one-dimensional dynamic array container (depicted in Figure 9). The MUL is used to calculate the item offset of the latest struct *Checkpoint* from the first one is *(length-1)* $\times$ *sizeof (Checkpoint)* and $SHA3_2$ is used to get array identity, which suggests the location that stores the array length. By inspecting how EVM stores an array, we reveal that the identity refers to a storage location that stores the length of the array [8]. Array items are stored contiguously, and the location of the first array item is the result of a $SHA3$ operation on

Nr.	Instruction	Nr.	Instruction	Nr.	Instruction	Nr.	Instruction
01	PUSH id	17	PUSH 00	33	MSTORE	49	PUSH 299
02	PUSH 00	18	SHA3$_1$	34	PUSH 20	50	JUMPI
03	DUP4	19	PUSH id	35	ADD	51	INVALID
04	PUSH 0xFF	20	DUP1	36	PUSH 00	52	JUMPDEST
05	AND	21	PUSH 00	37	SHA3$_3$	53	SWAP1
06	PUSH 0xFF	22	DUP6	38	DUP1	54	PUSH 00
07	AND	23	PUSH 0xFF	39	SLOAD	55	MSTORE
08	DUP2	24	AND	40	SWAP1	56	PUSH 20
09	MSTORE	25	PUSH 0xFF	41	POP	57	PUSH 00
10	PUSH 20	26	AND	42	SUB	58	SHA3$_2$
11	ADD	27	DUP2	43	DUP2	59	SWAP1
12	SWAP1	28	MSTORE	44	SLOAD	60	PUSH 02
13	DUP2	29	PUSH 20	45	DUP2	61	MUL
14	MSTORE	30	ADD	46	LT	62	ADD$_1$
15	PUSH 20	31	SWAP1	47	ISZERO	63	PUSH 01
16	ADD	32	DUP2	48	ISZERO	64	ADD$_2$

Fig. 16. Disassemble text for BabyDTXToken to locate bookkeeping, the attached number specifying the execution order of opcode.

the array's identity [8]. The two trees with the root $SHA3_1$ and $SHA3_2$ correspond to the mapping container (depicted in Figure 8) are used to access the variable *balances* (Line 8).

4.7.2 Searching Bytecode for Instruction Sequences of BabyDTXToken. In this section, we explain how `TokenAware` searches for bookkeeping in the bytecode b of the BabyDTXToken. First, `TokenAware` inspects the implementation of standard ERC20 interfaces and finds that b has implemented the transfer() interface (in Figure 14, Line 11). Then, `TokenAware` will check whether b contains an instruction sequence p that belongs to an item of containers ap. In Figure 16, we display the specific instruction sequence of ap and the highlighted gray instructions correspond to the opcodes of the tree format in Figure 15. By merging all instruction sequences in ap into three trees whose roots are the instruction sequences of three basic containers, we find a path in such tress. As shown in Figure 15, these paths are marked by dashed lines, indicating the combination between the basic containers. After obtaining the combination of the basic containers, we can conclude that b contains an instruction sequence p that belongs to ap. Finally, we use decompiler [30] to identify the code region executable attribute and count the execution number of the code blocks containing p. From the source code of BabyDTXToken, we can see that the transfer() of BabyDTXToken is public and it updates the bookkeeping twice. Therefore, we consider p of b as the instruction sequence to locate an item of the bookkeeping.

4.7.3 Recognizing Bookkeeping from Instructions of BabyDTXToken. In this step, `TokenAware` parses the instructions sequence p of BabyDTXToken to recognize bookkeeping. Specifically, `TokenAware` will recognize bookkeeping of BabyDTXToken by monitoring the execution of EVM instructions. Since we recognize the BabyDTXToken's container combination in Section 4.7.1 and the basic container in Section 4.2, when `TokenAware` encounters an instruction sequence shown in Figure 16, it can directly locate the basic containers in the instruction sequence and obtain the associated container combinations.

5 OPTIMIZATIONS

By fully exploiting unique features of Ethereum and the modern multi-core architecture, we propose four optimization mechanisms to improve the efficiency of TokenAware.

O1: cache. O1 leverages the unique feature of Ethereum that *code clone is prevalent in smart contracts* [31, 37]. Specifically, TokenAware computes the hash of the bytecode of the token smart contract that has been processed. Given the bytecode of a token smart contract, TokenAware queries the cache to check whether or not the same bytecode has been processed. If so, then TokenAware skips it; otherwise, TokenAware processes it to recognize bookkeeping variables and records the result in the cache. Section 6.3 shows that the overhead of TokenAware increases from 4% to 44% if O1 is turned off.

O2: parallelization. O2 is based on the feature that *Ethereum node does not fully use the multi-CPU and multi-core architecture of modern computers*, and there are substantial unused computing resources. TokenAware makes full use of the spare computing resources by parallelization. Specifically, TokenAware prepares a worker pool consisting of multiple workers, each of which binds to a thread. The current implementation of TokenAware has 20 workers. Besides, TokenAware considers bookkeeping recognition of each bytecode as a task, and uses one worker to execute a task. If all workers are busy, then tasks in the task queue should wait for available workers. If a worker finishes a task, then it will pick up a task from the task queue. Section 6.3 shows that the overhead of TokenAware increases from 4% to 26% if O2 is turned off.

O3: early recognition. O3 exploits two unique features of Ethereum. First, an Ethereum node downloads a number of blocks from other nodes, and then runs transactions in these blocks. Hence, *the execution of transactions falls behind the downloading of transactions*, i.e., an Ethereum node maintains a list of transactions waiting for execution. Second, *an external transaction for deploying a smart contract carries the bytecode of the contract*, and thus we can obtain the bytecode before executing the external transaction. O3 monitors the process of downloading blocks, and extracts the bytecode of smart contracts to be deployed by the external transactions. After that, TokenAware appends the bytecode to the task queue for bookkeeping recognition. O3 can reduce the overhead of TokenAware, because bookkeeping recognition is running when transactions are waiting for execution. Section 6.3 shows that the overhead of TokenAware increases from 4% to 17% if O3 is turned off.

O4: pipeline. O4 leverages the fact that a smart contract should be deployed before it can be invoked. Hence, the unique feature is that *there is usually a time gap between the time when deploying a smart contract and the time when the smart contract is invoked for the first time.* Given a token smart contract A, if the time gap exists, TokenAware executes the subsequent transactions, which do not invoke A in parallel with the bookkeeping recognition of A. Otherwise, TokenAware pauses the execution of A until the bookkeeping recognition of A is finished. O4 can reduce the overhead of TokenAware, because the time gap is utilized to run transactions. Section 6.3 shows that the overhead of TokenAware increases from 4% to 15%, if O4 is turned off.

Optimization combinations. Individual optimization techniques as well as their combinations can be applied to TokenAware. There are 16 ($\sum_{i=0}^{4} C_4^i$) combinations in theory, however, not all combinations are meaningful. Specifically, if we turn off O3 and O4, then it is meaningless to turn on O2. The reason is that without O3 and O4, TokenAware can only obtain the bytecode of a smart contract A when executing the transaction for deploying it and the subsequent transactions must wait for the finish of recognizing A's bookkeeping. Hence, turning on parallelization is meaningless, because there is at most one task at any time. Therefore, the optimization combination <O1 on, O2 on, O3 off, O4 off> is equivalent to <O1 on, O2 off, O3 off, O4 off>, and <O1 off, O2 on, O3 off, O4 off> is equivalent to <O1 off, O2 off, O3 off, O4 off>.

```
1 mapping (address => uint256) userID;
2 mapping (uint256 => uint256) balances;
3 function updateBalance(address _addr, uint256 _amount) {
4     uint id = userID[_addr];
5     balances[id] = _amount;
6 }
```

(a) Example of New Type 1

```
1 struct asset {
2     string name;
3     uint amount;
4 }
5 mapping(address => asset)
6   public balances;
```

(b) Example of New Type 2

Fig. 17. Examples of two new types.

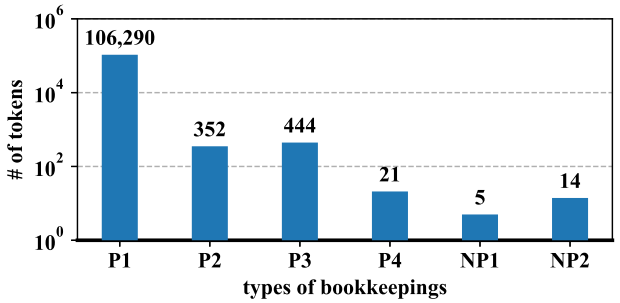


Fig. 18. Number of tokens using each bookkeeping type.

6 EVALUATION

We evaluate TokenAware with real transaction data by answering three research questions. All experiments are conducted on a server equipped with 1 Intel XEON CPU with 64 cores, 24 GB main memory, and 1 TB hard disk.

RQ1: How is the accuracy of TokenAware?

RQ2: How is the efficiency of TokenAware?

RQ3: Is each optimization mechanism effective in reducing the overhead?

6.1 Accuracy

We run TokenAware to process all transactions in 9 million blocks, and the nine millionth block was mined on November 25, 2019. We find that 19,678,790 smart contracts are deployed in total containing 246,476 unique bytecode (duplicated bytecode is eliminated). TokenAware recognizes 107,202 token smart contracts with unique bytecode from them. From all detected token smart contracts, TokenAware discovers six types of bookkeeping, including the four types adopted by TokenScope [8], and two new types. Figure 17 shows code snippets for two real token smart contracts that use these two new types of bookkeeping. Figures 10 and 11, respectively, illustrate the instruction sequence for locating the items of the two bookkeeping, because the two bookkeeping adopt the two modes of container combinations (C1 and C2), respectively. Since TokenScope relies on four manually defined types [8], which do not include these two new types, it cannot recognize the bookkeeping variables in these two token smart contracts. In contrast, TokenAware successfully recognizes them, because it is not limited to the previously known types. Figure 18 depicts the number of token smart contracts using each type of bookkeeping, where P1 to P4 are known types adopted by TokenScope [8] and NP1, NP2 are the two new types.

The *accuracy* of TokenAware is the proportion of token smart contracts whose token transfer behaviors are correctly recognized to the total token smart contracts. A token transfer behavior is *correctly* recognized, if the addresses of its token senders and receivers, and the token amounts

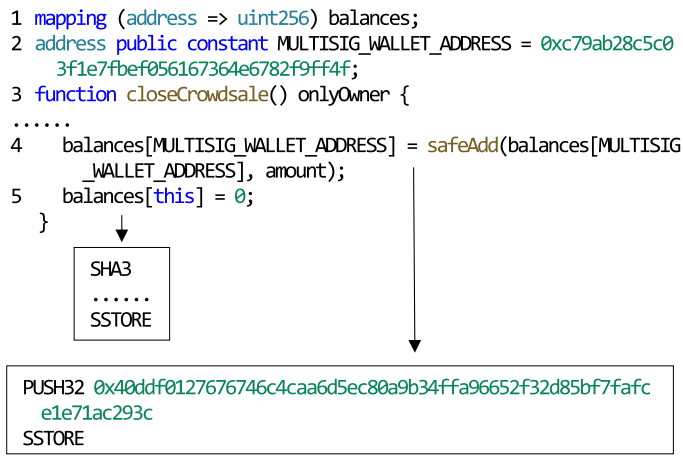


Fig. 19. A token transfer behavior that is not recognized by TokenAware accurately.

changed of all token senders and receivers are correct. TokenAware can recognize all token transfer behaviors of a token smart contract accurately, if both the two requirements meet: (1) its bookkeeping is correctly identified, and (2) all instruction sequences to locate an item of the bookkeeping are identified.

We cannot evaluate the accuracy of TokenAware on all 107,202 detected token bytecode due to the large number. Instead, we download the list of the most popular ERC20 tokens, which are validated by Etherscan [24]. The list contains 926 token smart contracts with their source code. By reading the source code of these 926 tokens, we find that TokenAware precisely recognizes the token transfer behaviors of 914 token smart contracts. Hence, the accuracy of TokenAware is 98.7% (914/926). For the remaining 12 token smart contracts, the second requirement is not satisfied although the bookkeeping of them are correctly recognized. Particularly, they are compiled with optimizations turning on and they access some of their bookkeeping items with hardcoded addresses. In this case, the locations of the accessed bookkeeping items are not computed at runtime; instead, these locations are pre-computed by the compiler, i.e., these locations are constants in the EVM bytecode, so TokenAware is not aware that these locations record bookkeeping items.

A token smart contract whose token transfer behaviors are not accurately recognized by TokenAware is shown in Figure 19. The bytecode snippets for modifying the token amounts of *MULTISIG_WALLET_ADDRESS* and the token smart contract itself (i.e., *this*) are also depicted in this figure. Its bookkeeping (Line 1) is correctly recognized from other functions of this token smart contract (not shown in this figure), however, the token transfer behavior incurred by executing closeCrowdsale() (Line 3) is not recognized accurately due to a hardcoded address (Line 2). More precisely, since the address of account *MULTISIG_WALLET_ADDRESS* is defined as a constant in this token smart contract and the instruction sequence used to locate an item of the bookkeeping is definite, the location for recording the token amount possessed by this account is fixed (i.e., 0x40...3c). To improve the performance of the smart contract, the compiler computes this location at compile time and hard encodes it into the bytecode of the contract. Therefore, the compiler does not generate the instruction sequence executed at runtime for locating the location 0x40...3c. Since TokenAware infers the token transfer behavior by monitoring the execution of such instruction sequence, it fails to detect the token increase of the account *MULTISIG_WALLET_ADDRESS*. On the contrary, TokenAware identifies that the token amount of *this* decreases to zero, because the location corresponding to *this* is computed at runtime. The imprecision caused by hardcoded

```
1  function transfer(address _to, uint256 _value) return (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;}
9  }
```

Fig. 20. A token smart contract that repeatedly emits Transfer events.

addresses is a difficult problem for the techniques based on the monitoring of bookkeeping, because the location of the modified bookkeeping item is a constant number that is difficult to be differentiated from other variables. However, such problem is not a big limitation of applying TokenAware in practice, because we find that only 1.3% of token smart contracts use hardcoded addresses to access bookkeeping items.

We thank the authors of TokenScope for providing their tool so that we can evaluate the accuracy of TokenScope on the set of 926 validated token smart contracts. We find that it processes the same result with TokenAware, so it has high accuracy in recognizing token transfers of the existing token smart contracts. We then compare the ability of TokenAware and TokenScope in coping with the evolution of smart contracts. To conduct such comparison, we construct all 251,016 ($f_0 + f_1 + f_2 + f_3$) containers whose combination operations are no larger than three. Then, we randomly select 100 containers out of them, and write 100 token smart contracts in Solidity for each container, each of which implements a function to transfer tokens by modifying the container. After that, we compile these token smart contracts into EVM bytecode. We run TokenAware and TokenScope to process these bytecode. Results show that TokenAware recognizes the bookkeeping variables in *all* of these 100 token smart contracts, but TokenScope recognizes *none*, because the four types of bookkeeping that can be identified by TokenScope are not included in 100 randomly selected bookkeeping. Therefore, TokenAware can cope with the evolution of smart contracts, because it captures the essence of how bookkeepings are constructed, but TokenScope cannot, because it depends on four fixed types of bookkeeping. Moreover, considering the tendency of malicious token smart contracts that will change the type of bookkeeping to evade the detection of TokenScope, TokenAware will be more accurate and robust in dealing with malicious token smart contracts.

We then compute the accuracy of the approaches based on standard interfaces and the accuracy of the approaches based on standard events. The former is the proportion of token smart contracts whose token transfer behaviors can be inferred correctly from standard interfaces, and the latter is the proportion of token smart contracts whose token transfer behaviors can be inferred correctly from standard events. We also evaluate these two accuracies with the 926 validated token smart contracts, and we find that the accuracies are just 4.5% and 35.6%, respectively. The reason for their low accuracies is that standard interfaces and standard events do not necessarily reflect real token transfer behaviors, especially that the majority of tokens use custom interfaces to transfer tokens and such token transfer behaviors cannot be inferred from standard interfaces.

Figure 20 illustrates a real token smart contract whose token transfer behavior is not accurately identified by monitoring standard events. Specifically, this token smart contract incorrectly implements the ERC20 token standard, i.e., it emits two Transfer events for each invocation of function transfer() (Lines 2 and 6). Thus, the approach based on standard event recognizes that account *msg.sender* transfers $2 \times$ *_value* tokens to account *_to*, because it recognizes the token transfer

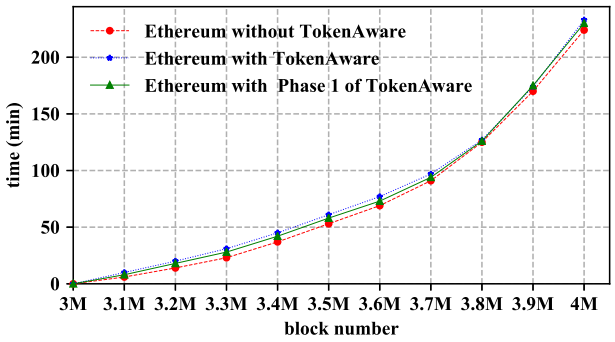


Fig. 21. Time used for processing blocks.

behavior by monitoring standard events. However, by observing the modifications on the bookkeeping, we can see that only *_value* tokens are transferred (Lines 4 and 5).

Answer to RQ1: The accuracy of TokenAware is 98.7%.

6.2 Efficiency

We first define the *overhead* of TokenAware. Assuming the time for running an Ethereum node without incorporating TokenAware is t_1 and the time for running the Ethereum node containing TokenAware is t_2, so the overhead is computed as $(t_2 - t_1)/t_1$. We perform the experiment when the Ethereum node downloads the blocks within the block interval [3,000,000, 4,000,000]. To obtain a reliable result, we repeat the experiment five times and report the average values. We do not conduct the experiment when the Ethereum node downloads the blocks within the block interval [0, 3,000,000) due to a notorious DoS attack happened within that block interval [5]. The DoS attack significantly slows down the Ethereum node, i.e., t_1 is very large, so the overhead of TokenAware will be over-optimistically low if we test TokenAware during the block interval [0, 3,000,000). We will test TokenAware in a larger block interval in future.

Figure 21 shows that an Ethereum node without TokenAware costs 224 min to process the blocks within the block interval [3,000,000, 4,000,000], and the Ethereum node equipped with TokenAware costs 233 min to process the same blocks. Therefore, the overhead of TokenAware is 4% ((233 − 224)/224). Please recall that TokenAware consists of two phases, one for bookkeeping recognition and one for inferring token transfer behaviors. By turning off the second phase, we measure the time for bookkeeping recognition, as shown in Figure 21. Results show that bookkeeping recognition incurs 2.7% overhead, which accounts for 68% of the overall overhead.

Figure 22 depicts the **cumulative distribution function (CDF)** plot of bookkeeping recognition time. Each point (x, y) in Figure 22 means that for y of smart contracts, TokenAware costs no longer than xs for bookkeeping recognition from each of them. The time for bookkeeping recognition ranges from 0.2 s to 19 s, and for 90% of smart contracts, TokenAware costs no longer than 0.97 s. We then find three factors that affect the time for bookkeeping recognition. The first is the length of bytecode, because our approach searches the bytecode for the instruction sequences for locating an item of bookkeeping variables (step 2, Section 4.4). The second is the complexity of the EVM instructions contained in the bytecode, because our approach interprets the instructions to obtain operands and results (step 2, Section 4.4). The third factor is the decompiler, which is used for identifying the code within each public/external function (step 3, Section 4.4).

We then evaluate the overhead of TokenScope, an offline tool that records the traces for all executions of smart contracts and infers token transfer behaviors by analyzing the traces. Our

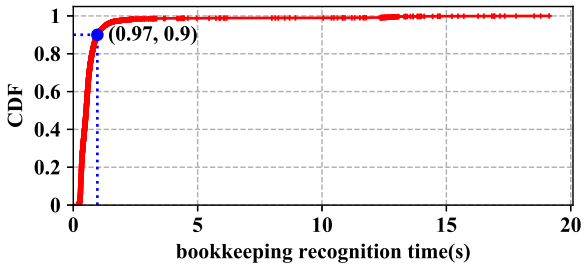


Fig. 22. CDF of bookkeeping recognition time.

experiment discovers that TokenScope costs 3,120 min to process the blocks within the interval [3,000,000, 4,000,000], and thus its overhead is 12.9, which is 323× the overhead incurred by TokenAware. Hence, TokenScope is not suitable to online application scenarios. As described in Section 3.3, given the high overhead of TokenScope, it cannot identify token transfers in a transaction before it is confirmed. In comparison, TokenAware takes only 0.52 s (4% overhead × 13 s for mining a new block), which is much less than the time required by TokenScope. Therefore, TokenAware can facilitate the detection of malicious behaviors relevant to tokens, because it makes it possible to take countermeasures before transactions involving malicious behaviors are confirmed.

After that, we evaluate the overhead incurred by the techniques based on standard interfaces and standard events. To do so, we modify an Ethereum node so that it firstly monitors the function invocations and event emissions, then checks whether standard interfaces are invoked and whether standard events are emitted by parsing transaction data and instruction operands, and finally obtain token transfer behaviors by parsing the parameters of standard interfaces and standard events. Our experiment reveals that the technique relying on standard interfaces and standard events costs 227 min to process the blocks within the interval [3,000,000, 4,000,000], and thus its overhead is 1.3%, which is comparable to our TokenAware.

Answer to RQ2: The overhead of TokenAware is 4%.

6.3 Effectiveness of Optimizations

To validate the effectiveness of each optimization technique, we run an Ethereum node, in which TokenAware is implemented with each of 14 optimization combinations, as shown in Table 1, to download the blocks within the block interval [3,000,000, 4,000,000]. The reason for selecting such block interval is explained in Section 6.2. To obtain a reliable result, we repeat the experiment five times and report the average values. The overall time consumption for the experiments is about 1,450 h, and we plan to conduct the experiments in a larger block interval in our future work. The overhead of each optimization combination is shown in Table 1. The columns O1, O2, O3, and O4 indicate four optimization techniques. The √ and × mean that an optimization technique is applied and not applied, respectively. We have many insights from the experimental results.

—***Insight 1:*** *Applying all optimizations achieves the best performance.* The overhead of COMB14 is 4%, which is 1/345 (4%/13.8) of the overhead incurred by COMB1 with no optimizations.

—***Insight 2:*** *Applying more optimizations achieves better performance.* Figure 23 presents the overhead of all optimization combinations, where the x-axis stands for the number of optimizations and the y-axis indicates the overhead. We can see that the overhead of applying all four optimizations is about 4/15 (4%/15%), 1/11 (4%/44%), and 1/13 (4%/52%) of the overheads incurred by applying the best of three optimizations (i.e., COMB10), the best of two optimizations (i.e., COMB6) and the best of one optimization (i.e., COMB2), respectively.

Table 1. Overhead of all Optimization Combinations

	O1	O2	O3	O4	Overhead
COMB1	×	×	×	×	13.8
COMB2	✓	×	×	×	52%
COMB3	×	×	✓	×	13.8
COMB4	×	×	×	✓	12.6
COMB5	✓	×	✓	×	50%
COMB6	✓	×	×	✓	44%
COMB7	×	✓	✓	×	8.8
COMB8	×	✓	×	✓	60%
COMB9	×	×	✓	✓	12.6
COMB10	✓	✓	✓	×	15%
COMB11	✓	×	✓	✓	26%
COMB12	✓	✓	×	✓	17%
COMB13	×	✓	✓	✓	44%
COMB14	✓	✓	✓	✓	4%

O1: cache O2: parallelization.
O3: early recognition O4: pipeline.

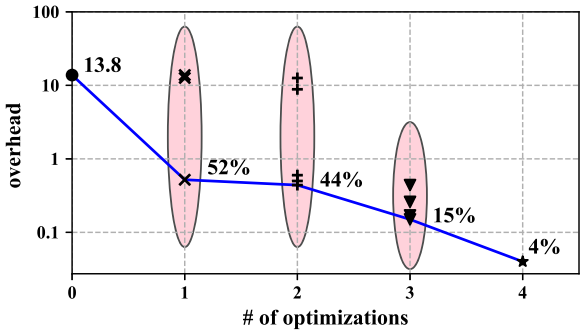


Fig. 23. Overhead of different number of optimizations.

—***Insight 3:*** *Applying cache only can achieve good performance.* The overhead of COMB2 is 52%, which is about 1/27 (52%/13.8) of the overhead incurred by COMB1. Moreover, the overhead of applying cache only is comparable to the overhead of applying all other optimizations (i.e., the overhead of COMB13 is 44%). After carefully investigating our experimental results, we find the reason to explain the good performance of cache. Particularly, the bytecode of 661,933 smart contracts is repeated, so `TokenAware` just needs to analyze 9,179 bytecode, which only accounts for 1.4% (9,179/(9,179 + 661,933)) of all smart contracts. We can see from Figure 24 that the cache hit ratio increases from 54.6% to 98.6% along with the increasing of the block number. Figure 24 also reveals that the overhead of `TokenAware` decreases from 66.6% to 4%, which is roughly in pace with the increasing of the cache hit ratio. We also notice that the overhead of processing the blocks within [3,000,000, 4,000,000] is higher than the overhead of processing the blocks within [3,000,000, 3,800,000]. By checking the experimental results, we find the reason that more bytecode that should be analyzed will incur higher overhead. Particularly, 2,036 new unique bytecode is deployed within the block interval [3,800,000, 4,000,000], which is 75% ((2,036 − 1,1161)/1,161) more than the new

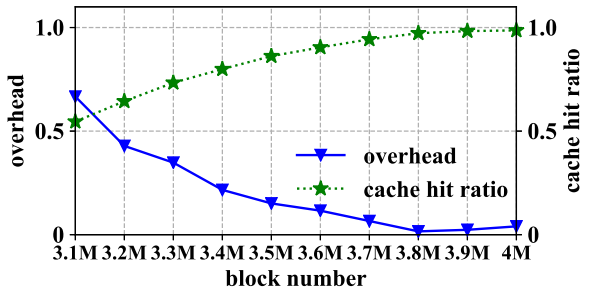


Fig. 24. Overhead and cache ratio.

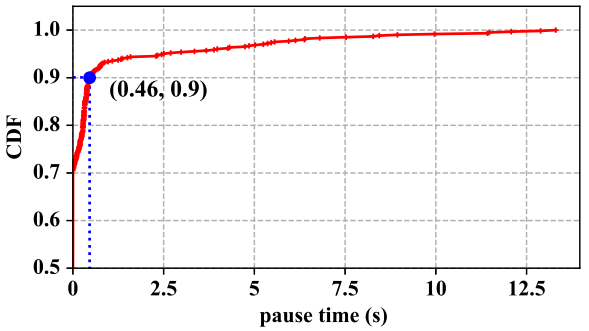


Fig. 25. CDF of pause time.

unique bytecode deployed within the block interval [3,600,000, 3,800,000), and thus the overhead increases when processing the blocks from 3,800,000 to 4,000,000.

—***Insight 4:*** *Pipeline is effective in reducing the overhead.* For example, the overhead of the `TokenAware` with full optimizations (i.e., COMB14) reduces the overhead of the `TokenAware` without pipeline (i.e., COMB10) from 15% to 4%. We then investigate why pipeline is effective by evaluating the pause time of executed smart contracts. Please recall that `TokenAware` pauses the execution of a smart contract at its first instruction, if `TokenAware` has not finished bookkeeping recognition of that smart contract yet. Among all 9,179 executed smart contracts, the pause time of 8,575, which accounts for 93.4%, is zero, indicating that for 93.4% of smart contracts, `TokenAware` finishes bookkeeping recognition before they are invoked for the first time.

For the remaining 6.6% smart contracts, their pause time ranges from 1 ms to 13.3 s, as shown in Figure 25. A point (x, y) in Figure 25 means that for y smart contracts, the pause time for each of them is no longer than xs. We observe that for 90% of smart contracts, the pause time for each of them is no longer than 0.46 s. The small proportion of smart contracts that need to pause and the short pause time are the reasons for the effectiveness of pipeline.

—***Insight 5:*** *The number of workers significantly impacts the overhead.* To evaluate the impact caused by the degree of parallelism, we run the Ethereum node equipped with `TokenAware` with varied workers, ranging from 1 to 30. Please note that the worker number is 1 indicating that parallelization is turned off. Figure 26 suggests that the overhead decreases with the increasing of the worker number. Besides, the overhead achieves 4% when the worker pool has 15 workers, and the overhead will not decrease with more workers. How many workers are needed to achieve the best performance depends on two factors. The first is the number of tasks, which is determined by the practical blockchain data. More tasks require more workers. The second is the efficiency of bookkeeping recognition. Faster bookkeeping recognition requires fewer workers. Interestingly, we

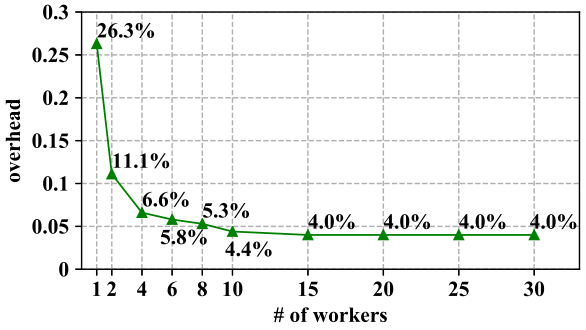


Fig. 26. Overhead of varied workers.

find that the overhead of two workers (i.e., 11.1%) is lower than half of the overhead of one worker (i.e., 26.3%). Such result does not mean that two workers achieve >2× speedup in bookkeeping recognition. Instead, the reason is that any optimizations for accelerating phase 1 (i.e., bookkeeping recognition) can also speed up phase 2 (i.e., inferring token transfer behaviors), because faster bookkeeping recognition leads to shorter pause time in executing token smart contracts.

—***Insight 6:*** *Either cache or parallelization is a necessary optimization to reduce the overhead to below 100%.* On the contrary, by just applying early recognition and pipeline (i.e., COMB9), the overhead is 12.6, which is faster than COMB1 by merely 8.7% ((13.8 − 12.6)/13.8). We present the reason as follows. Without cache, many smart contracts with the same bytecode should be analyzed, and therefore there should exist too many tasks in the task queue. Consequently, these tasks may wait for execution, because there is only one worker when parallelization is turned off.

—***Insight 7:*** *Applying both cache and early extraction does not obviously outperform applying cache only.* Compared to applying cache only, the overhead just decreases by 3.8% ((52%−50%)/52%) after adding early recognition. We find the reason that although COMB5 generates tasks quickly than COMB2 due to early recognition, some tasks may wait for execution, because there is only one worker available by turning off parallelization.

—***Insight 8:*** *Parallelization significantly reduces the overhead, if there are abundant tasks.* By adding parallelization, the overhead of COMB8 is 4.8% (60%/12.6) of the overhead led by COMB4. The reason is that pipeline can quickly generate many tasks, if there are many transactions for deploying smart contracts, which are enabled to run without waiting for the results of bookkeeping recognition when pipeline is turned on. Similarly, by adding parallelization, the overhead of COMB7 is 64% (8.8/13.8) of the overhead incurred by COMB3. Please recall that early recognition scans all transactions in all downloaded blocks to extract the bytecode that should be deployed by external transactions. Our experiment reveals that such process for handling the blocks within the interval [3,000,000, 4,000,000] just needs 2 min, so early recognition can generate many tasks quickly.

Answer to RQ3: All proposed optimization techniques are effective in reducing the overhead.

7 AN APPLICATION BASED ON `TOKENAWARE`

We develop an application based on `TokenAware` to demonstrate how `TokenAware` can be used to detect a new type of malicious behavior relevant to tokens, so-called *fraudulent transfer*. Fraudulent transfer means that the attacker exploits a vulnerability in the **decentralized application (DApp)**

Fig. 27. The function call relevant to the attack and the code snippet containing the vulnerability in Qubit Finance.

to create a fake token transfer. However, it does not really transfer the tokens, thus cheating the DApp. The vulnerability in the DApp means that the DApp incorrectly sends a function call to an EOA instead of a token smart contract.

We explain the mechanism of fraudulent transfers through a recent real-world case. On Jan. 28, 2022, attackers stole more than $80 million worth of assets from Qubit Finance, a **decentralized finance (DeFi)** application that provides lending services, using fraudulent transfers [38]. This incident is the seventh largest attack on the DeFi application according to the amount of funds stolen [38]. Figure 27 shows the victim's code snippet containing the vulnerability. The function safeTransferFrom() serves to transfer the collateral from the borrower to Qubit Finance so that Qubit Finance can subsequently provide a loan to the borrower based on the value of the collateral. The variable *token* (Line 2) indicates the token used as collateral, and the function safeTransferFrom() invokes the ERC20 standard method transferFrom() of the token smart contract by a low-level call to transfer tokens (Line 8). Please note that the function id 0x2378b2dd indicates the ERC20 standard method transferFrom(). However, the low-level call has a bug that does not check whether the target of the call is a smart contract or not. Note that when the target of a low-level call is an EOA or a zero address (i.e., 0x0000000000000000000000000000000000000000), EVM does not conduct any operation but returns *true* directly [26]. The so-called target refers to the account to which the function call is sent. The attacker exploited the vulnerability by setting the parameter *token* to a zero address when transferring collateral to Qubit Finance (shown in the upper part of Figure 27), which misled Qubit Finance to incorrectly identify that the collateral was successfully transferred, because it received *true* returned by the low-level call, however, no tokens were actually transferred. Eventually, the attacker falsified a large amount of collateral and was allowed to lend out assets.

By manually analyzing the transactions involved in this attack, we summarize four characteristics of fraudulent transfers. Hence, we develop a new application based on TokenAware to uncover fraudulent transfers by using the following four rules to detect the relevant transactions. **Rule 1:** The ERC20 standard method (i.e., transfer() or transferFrom()) should be called and the target of the call should be an EOA or zero address. This rule is used to characterize the first step of the attack. That is, the DApp incorrectly sends a function call to an EOA or zero address instead of a token smart contract. **Rule 2:** The sender and receiver of the token should not be the same, and the number of tokens is greater than zero (i.e., $from \neq to$ and $value > 0$). Rule 2 describes the expected token transfer behavior. That is, some tokens should be transferred between different accounts. **Rule 3:** there is not any operation on the bookkeeping. That is, no token is transferred. Rule 3 indicates that the tokens do not transfer as expected. **Rule 4:** the return value of the EVM low-level call should be *true*, which can mislead the DApp. This rule indicates that the DApp incorrectly identifies a successful token transfer, because it receives *true* returned by the low-level call.

We use the example shown in Figure 27 to describe the detection steps. First, with TokenAware's ability to monitor the execution of smart contracts (Section 3.2), our application records any external function call that is generated during the execution of safeTransferFrom() and invokes the ERC20 standard interface transferFrom() (Line 8) with the target equal to zero address. As shown in the lower half of Figure 27, TokenAware detects that the execution of codes in line 8 generates an external function call. That is, transferFrom() is called, and the target of the call equal to zero address. Second, after parsing the parameters of the function transferFrom(), we find that *from* (i.e., 0xD01Ae1A708614948B2B5e0B7AB5be6AFA01325c7 in Figure 27) is not equal to *to* (i.e., 0x17B7163cf1Dbd286E262ddc68b553D899B93f526 in Figure 27) and *value* (i.e., $190.000000 \times 10^{18}$ in Figure 27) is greater than zero. Third, TokenAware does not observe any operation on the bookkeeping during the execution of transferFrom(), meaning that no token is transferred. Fourth, our application detects that the return value of the EVM low-level call is *true* when the execution of transferFrom() is finished. Finally, since all four rules are satisfied, we detect the fraudulent transfer.

To evaluate this application, we replay all transactions from the launching of Ethereum (Jul. 30, 2015) to Jan. 31, 2022 by node synchronization and use this application to detect fraudulent transfers.

Using TokenAware to detect fraudulent transfer, we discovered 25 fraudulent transfers from 1,936,572 transactions to the contracts that were under attack since July 2015. Note that the 25 transactions contain the reported attack described above [38]. Although 25 is not a large value, one possible reason is that the majority of users are benign and it is not easy for users to identify the exploitable issues in the bytecode of smart contracts without the assistance of tools like TokenAware. Besides, we also confirm that no false positives are produced by manually analyzing these transactions and smart contracts.

Furthermore, we observe that a popular decentralized exchange market called 0x protocol, which offers the token swap service to its users [1], was also suffering from fraudulent transfers. As shown in Figure 28 (please ignore the highlighted parts now), to swap one type of token for another, the user (also called *Maker*) of the 0x protocol first creates an order that specifies the token (i.e., *makerAsset*) to be spent, the token (i.e., *takerAsset*) to be received, the price of the two tokens. Then, if there exists another user (also called *Taker*) who has a need to exchange *takerAsset* for *makerAsset*, it can send a transaction to the 0x protocol's smart contract to declare that it agrees to trade tokens in accordance with this order. Finally, 0x protocol enables tokens to be transferred between *Maker* and *Taker* by invoking the standard interface of both token smart contracts.

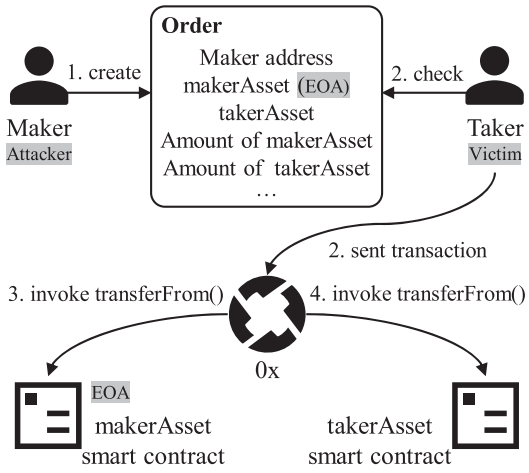


Fig. 28. Workflow of 0x protocol.

In the attack we detected (shown in Figure 28), the attacker first created a malicious order in which the *makerAsset* points to an EOA and the *takerAsset* points to a normal token smart contract. Moreover, the information in the order indicates that *Taker* can pay only one *takerAsset* token in exchange for a large number of *makerAsset* tokens. After that, the victimized *Taker* agreed to the token exchange without carefully checking the order. Then, 0x protocol invoked the function transferFrom() of *makerAsset* and *takerAsset*, respectively, to enable the transfer of tokens. However, since *makerAsset* points to an EOA, the fraudulent transfer occurs, resulting in the victimized *Taker* paying out a token without receiving any token in exchange.

We will develop more applications based on `TokenAware` to detect other malicious behaviors relevant to tokens, such as facilitating the detection of money laundering through cryptocurrencies, in future work. Specifically, money laundering through cryptocurrencies refers to keeping illegally obtained assets untraceable by exploiting the anonymity of cryptocurrencies and the fact that cryptocurrencies are difficult to regulate by law [39]. To prevent the incriminated funds from being traced to their source, money launderers convert funds into cryptocurrencies and send them back and forth between different crypto wallets [48]. The process of money laundering involves a large number of token transfers, and `TokenAware` can facilitate the detection of money laundering through cryptocurrencies as it can recognize the token transfer behavior in real time and accurately.

8 THREATS TO VALIDITY

We have manifested the validity threats of our research in the following aspects.

Internal validity. For the internal validity, we consider the effect of hardcoded address and inline assembly.

(1) hardcoded address. When a contract is compiled with optimization, its bytecode might access some of their bookkeeping items through hardcoded address. In this case, the locations of the accessed bookkeeping items are not computed at runtime; instead, these locations are pre-computed by the compiler and become constants in the EVM bytecode. Therefore, `TokenAware` is not aware that these locations refer to bookkeeping items.

Fortunately, we obverse that only 1.3%(12/926) of token smart contracts locate their bookkeeping items by hardcoded address on the most popular 926 token smart contracts.

(2) inline assembly. The access patterns (i.e., the basic containers and their combinations) captured by TokenAware are generated by the Solidity compiler. But developers might use inline assembly to construct their own access patterns of bookkeeping. Currently, TokenAware cannot recognize such types of bookkeeping. We recently conducted a large-scale empirical study on open-source smart contracts with inline assembly [36] and did not observe any token smart contact accessing the bookkeeping by the inline assembly.

External validity. The threats to the external validity include the limited scale of dataset and the lack of ground truth for fully evaluating false negatives.

We evaluate the accuracy of TokenAware by using 926 most popular ERC20 tokens on Section 6.1 and the efficiency of TokenAware by using the transactions in the 3,000,000th–4,000,000th blocks on Section 6.2. In future work, we will evaluate TokenAware by using a much larger dataset, and we will release the source code of TokenAware so that other researchers can evaluate it using various datasets.

We did not evaluate the false negatives of TokenAware due to the lack of dataset with ground truth. Since constructing such a dataset will take lots of time and manual effort, in future work, we will recruit many experienced developers to manually inspect smart contracts for building such a dataset with ground truth. Then, we will use it to evaluate the false negatives of TokenAware and contribute it to the community.

9 RELATED WORK

There are two categories of techniques for recognizing token transfer behaviors.

Techniques based on standard interfaces and standard events. Many tools used in practice belong to this category, including blockchain explorers (e.g., EthVM [17], ETCExplorer [18], BlockScout [45]), wallets (e.g., MetaMask [40], MyEtherWallet [41], Etherwall [25]), exchange markets (e.g., EtherEx [23], openANX [44]), and data analytic tools (e.g., Ethereum ETL [29]). Moreover, many academic studies belong to this category. DataEther is a data analytic platform that detects token transfer behaviors based on standard interfaces and standard events [7]. XBlock-ETH is a similar platform leveraging standard events [53].

A few studies organize token transfer behaviors as graphs, and then apply graph theory to measure these graphs [9, 47, 49]. Some empirical studies characterize some features (e.g., source code complexity) of Ethereum and smart contracts [12, 13, 43]. Dyson et al. design a digital forensics tool for recognizing and tracking token transfers [16]. These studies [9, 12, 13, 16, 43, 47, 49] rely on standard interfaces and standard events. Besides identifying token smart contracts, standard interfaces and standard events can be used for determining which token standards are followed by these token smart contracts [14, 15]. Although this category is widely applied, it suffers from low accuracy (35.6%), because the implementation of token smart contracts may not strictly follow the standards.

Techniques based on bookkeeping. The techniques of this category infer token transfer behaviors by monitoring the modifications to bookkeeping variables. Frowis et al. apply a heuristic rule to EVM bytecode by symbolic execution and taint analysis [28]. However, their method has high false negative rate due to its coarse-grained rules and the inherent disadvantages (e.g., path explosion, under-tainting) of symbolic execution and taint analysis [28]. TokenScope relies on four manually-identified bookkeeping types [8]. Manual bookkeeping recognition is time-consuming and may miss some bookkeeping types. Our experiments show that TokenScope misses two bookkeeping types used in practice, and it is not suitable for new smart contracts, which are continuously evolving. Moreover, TokenScope is an offline tool with significant overhead, and thus it is not suitable for online scenarios.

10 CONCLUSION

We propose and develop TokenAware, a novel online system for accurately and efficiently recognizing the bookkeeping and the operations on it for token smart contracts. TokenAware recognizes the bookkeeping by first learning the instruction sequences of basic containers and then deriving the instruction sequences of the combinations of basic containers. Moreover, it is equipped with four mechanisms for improving the performance. Extensive evaluation with real blockchain data demonstrates the high accuracy, high efficiency of TokenAware, and the effectiveness of all optimizations. Finally, the application based on TokenAware demonstrates how it facilitates malicious behavior detection.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments that greatly helped improve the presentation of this article.

REFERENCES

[1] 0x Protocol. 2022. 0x Documentation. Retrieved from https://docs.0x.org/introduction/welcome.
[2] Eric Banisadr. 2018. How $800k Evaporated from the PoWH Coin Ponzi Scheme Overnight. Retrieved from https://medium.com/@ebanisadr/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530.
[3] Rhonda Bush and Soohyun Choi. 2019. Forecasting Ethereum STORJ token prices: Comparative analyses of applied bitcoin models. In *Proceedings of the International Conference on Data Mining Workshops (ICDMW'19)*. IEEE, 216–223.
[4] Chainalysis. 2019. Why you should be watching ERC-20 Tokens. Retrieved from https://blog.chainalysis.com/reports/why-you-should-be-watching-erc-20-tokens.
[5] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. 2017. An adaptive gas cost mechanism for Ethereum to defend against under-priced dos attacks. In *Proceedings of the International Conference on Information Security Practice and Experience*. Springer, 3–24.
[6] Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li, et al. 2021. Sigrec: Automatic recovery of function signatures in smart contracts. *IEEE Transactions on Software Engineering* 48, 8 (2021), 3066–3086.
[7] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ang Chen, Kun Yang, Bin Hu, Tong Zhu, Shifang Deng, Teng Hu, et al. 2019. Dataether: Data exploration framework for Ethereum. In *Proceedings of the IEEE 39th International Conference on Distributed Computing Systems (ICDCS'19)*. IEEE, 1369–1380.
[8] 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 *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 1503–1520.
[9] Weili Chen, Tuo Zhang, Zhiguang Chen, Zibin Zheng, and Yutong Lu. 2020. Traveling the token world: A graph analysis of Ethereum ERC20 token ecosystem. In *Proceedings of the Web Conference*. 1411–1421.
[10] Yuzhou Chen and Hon Keung Tony Ng. 2019. Deep learning Ethereum token price prediction with network motif analysis. In *Proceedings of the International Conference on Data Mining Workshops (ICDMW'19)*. IEEE, 232–237.
[11] Zhen Cheng, Xinrui Hou, Runhuai Li, Yajin Zhou, Xiapu Luo, Jinku Li, and Kui Ren. 2019. Towards a first step to understand the cryptocurrency stealing attack on Ethereum. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19)*. 47–60.
[12] Monika Di Angelo and Gernot Salzer. 2020. Characteristics of wallet contracts on Ethereum. In *Proceedings of the 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS'20)*. IEEE, 232–239.
[13] Monika Di Angelo and Gernot Salzer. 2020. Characterizing types of smart contracts in the Ethereum landscape. In *Proceedings of the 4th Workshop on Trusted Smart Contracts, Financial Cryptography*.
[14] Monika Di Angelo and Gernot Salzer. 2020. Tokens, types, and standards: Identification and utilization in Ethereum. In *Proceedings of the International Conference Decentralized Applications and Infrastructures (DAPPS'20)*.
[15] Monika Di Angelo and Gernot Salzer. 2020. Towards the identification of security tokens on Ethereum. In *Proceedings of the 3rd International Workshop on Blockchains and Smart Contracts (BSC'20)*.
[16] Simon F. Dyson, William J. Buchanan, and Liam Bell. 2020. Scenario-based creation and digital investigation of Ethereum ERC20 tokens. *Forensic Sci. Int.: Dig. Invest.* 32 (2020), 200894.
[17] enkrypt. 2018. EthVM: Open Source Ethereum Blockchain Explorer. Retrieved from https://github.com/enKryptIO/ethvm.
[18] Ethereum. 2018. ETCExplorer. Retrieved from https://github.com/ethereumclassic/explorer.

[19] Ethereum. 2020. Solidity documentation. Retrieved from https://solidity.readthedocs.io/en/latest/.
[20] Ethereum. 2021. Ethereum Development Documentation. Retrieved from https://ethereum.org/en/developers/docs/blocks/#block-time.
[21] Ethereum. 2022. Ethereum Whitepaper. Retrieved from https://ethereum.org/en/whitepaper/.
[22] Ethereum. 2022. Types—Solidity 0.8.12 documentation. Retrieved from https://docs.soliditylang.org/en/v0.8.12/types.html#.
[23] EtherEx. 2018. EthEx: Decentralized exchange built on Ethereum. Retrieved from https://github.com/etherex/etherex.
[24] Etherscan. 2020. Token tracker—ERC20 tokens. Retrieved from https://etherscan.io/tokens.
[25] Etherwall. 2018. Etherwall: The first Ethereum desktop wallet. Retrieved from https://www.etherwall.com/.
[26] Stack Exchange. 2018. Solidity: Using low level call function on an EOA. Retrieved from https://ethereum.stackexchange.com/questions/56743/solidity-using-low-level-call-function-on-an-eoa.
[27] Python Software Foundation. 2022. Collections—Container datatypes. Retrieved from https://docs.python.org/3/library/collections.html.
[28] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. 2019. Detecting token systems on Ethereum. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 93–112.
[29] Google. 2019. Ethereum ETL. Retrieved from https://github.com/blockchain-etl/ethereum-etl.
[30] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, declarative decompilation of smart contracts. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 1176–1186.
[31] Ningyu He, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang. 2020. Characterizing code clones in the Ethereum smart contract ecosystem. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 654–675.
[32] Daniel Jennings. 2018. What Is Driving Ethereum's Coin Price Through The Roof? Retrieved from https://seekingalpha.com/instablog/22912651-daniel-jennings/5097449-what-is-driving-ethereum-s-coin-price-through-roof.
[33] Jerry. 2020. Step-by-Step Guide to Tokenizing Real-World Assets. Retrieved from https://theblockbox.io/step-by-step-guide-to-tokenizing-real-world-assets/.
[34] Ken Kennedy. 1978. Use-definition chains with applications. *Comput. Lang.* 3, 3 (1978), 163–179.
[35] Xiaoqi Li, Ting Chen, Xiapu Luo, Tao Zhang, Le Yu, and Zhou Xu. 2020. Stan: Towards describing bytecodes of smart contract. In *Proceedings of the 20th IEEE International Conference on Software Quality, Reliability, and Security*.
[36] Zhou Liao, Shuwei Song, Hang Zhu, Xiapu Luo, Zheyuan He, Renkai Jiang, Ting Chen, Jiachi Chen, Tao Zhang, and Xiao-song Zhang. 2022. Large-scale empirical study of inline assembly on 7.6 million Ethereum smart contracts. *IEEE Trans. Softw. Eng.* In Press.
[37] Han Liu, Zhiqiang Yang, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. 2019. Enabling clone detection for Ethereum via smart contract birthmarks. In *Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC'19)*. IEEE, 105–115.
[38] Shaurya Malwa. 2022. DeFi Protocol Qubit Finance Exploited for $80M. Retrieved from https://finance.yahoo.com/news/defi-protocol-qubit-finance-exploited-071509620.html.
[39] Jonathan T. Marks. 2022. Cryptocurrency and money laundering: why understanding fraud is critical. Retrieved from https://www.bakertilly.com/insights/cryptocurrency-and-money-laundering.
[40] METAMASK. 2018. METAMASK—Brings Ethereum to your browser. Retrieved from https://metamask.io/.
[41] MyEtherWallet. 2018. MyEtherWallet. Retrieved from https://www.myetherwallet.com/.
[42] The C++ Resources Network. 2022. Standard Containers. Retrieved from cplusplus.com/reference/stl/.
[43] Gustavo A. Oliva, Ahmed E. Hassan, and Zhen Ming Jack Jiang. 2020. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empir. Softw. Eng.* 25, 3 (2020), 1864–1904.
[44] openANX. 2017. openANX: Decentralised Exchange Token Sale Smart Contract. Retrieved from https://github.com/openanx/OpenANXToken.
[45] POA. 2018. BlockScout, Blockchain Explorer for inspecting and analyzing EVM Chains. Retrieved from https://github.com/poanetwork/blockscout.
[46] Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, and Ronan Sandford. 2018. EIP-1155: Multi Token Standard. Retrieved from https://eips.ethereum.org/EIPS/eip-1155.
[47] Shahar Somin, Goren Gordon, Alex Pentland, Erez Shmueli, and Yaniv Altshuler. 2020. ERC20 transactions over Ethereum blockchain: Network analysis and predictions. Retrieved from https://arXiv:2004.08201.
[48] Fabian Maximilian Johannes Teichmann and Marie-Christin Falker. 2021. Money laundering via cryptocurrencies–potential solutions from liechtenstein. *J. Money Launder. Control* 24, 1 (2021), 91–101.
[49] Friedhelm Victor and Bianca Katharina Lüders. 2019. Measuring Ethereum-based erc20 token networks. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 113–129.

[50] Fabian Vogelsteller and Vitalik Buterin. 2015. EIP-20: ERC-20 Token Standard. Retrieved from https://eips.ethereum.org/EIPS/eip-20.
[51] Gavin Wood. 2020. Ethereum: A Secure Decentralized Generalized Transaction Ledger. Retrieved from https://ethereum.github.io/yellowpaper/paper.pdf.
[52] ZeusTrade. 2018. Topic: there was a coin out of my wallet that I did not even get what it is. Retrieved from https://bitcointalk.org/index.php?topic=5023796.0.
[53] Peilin Zheng, Zibin Zheng, Jiajing Wu, and Hong-Ning Dai. 2020. Xblock-ETH: Extracting and exploring blockchain data from Ethereum. *IEEE Open J. Comput. Soc.* 1 (2020), 95–106.

Received 15 November 2021; revised 14 July 2022; accepted 22 July 2022