



DAppHunter: Identifying Inconsistent Behaviors of Blockchain-based Decentralized Applications

Jianfei Zhou[†]*University of Electronic Science and Technology of China*
Chengdu, China

Tianxing Jiang

University of Electronic Science and Technology of China
Chengdu, China

Haijun Wang

Ant Group
Hangzhou, China

Meng Wu

Ant Group
Hangzhou, ChinaTing Chen[✉]*University of Electronic Science and Technology of China*
Chengdu, China

Abstract—A blockchain-based decentralized application (DApp) refers to an application typically using web pages or mobile applications as the front-end and smart contracts as the back-end. The front-end of the DApp helps users generate transactions and send them to the user’s blockchain wallet. After the user signs and confirms the transaction using the blockchain wallet, the transaction will invoke the smart contract of the DApp. However, users bear the following risks when using DApps because of the potential inconsistent behaviors in DApps. First, the DApp front-end may generate incorrect transactions inconsistent with users’ intentions. Second, the smart contract may have misbehaviors when executing the transactions. Inconsistent behaviors of DApps not only lead to user confusion but also cause significant financial losses. In this paper, we proposed a novel approach to identify inconsistent behaviors of DApps on EVM-compatible blockchains by contrasting the behaviors of DApps that derived from the front-end, blockchain wallet, and smart contracts, respectively. We implemented our approach into a prototype named *DAppHunter*. We have applied *DAppHunter* on 92 real-world DApps of Ethereum and Binance Smart Chain and successfully identified 37 DApps with inconsistent behaviors. We confirmed that 35 of them are scam DApps and over 5 million blockchain addresses are at risk of becoming victims of these inconsistent DApps.

Index Terms—blockchain, smart contract, DApp testing, inconsistent behavior

I. INTRODUCTION

A blockchain consists of a growing list of records, called blocks, which are chained together using a cryptographic hash [1]. The underlying structure of a blockchain platform is a P2P overlay that consists of multiple nodes. In 2015, Ethereum was launched and quickly became one of the most popular blockchain platforms. Ethereum supports smart contracts, which are programs that are executed in the Ethereum Virtual Machine (EVM) [1]. With the rapid growth of smart contracts, blockchain-based decentralized applications (DApps) have developed rapidly. DApps have become more and more popular in the market. In 2021, the annual market value of transactions related to NFT trading DApps has reached up to \$22 billion. And the total value locked in DApps is measured

at \$189 billion [2]. A typical DApp is an application that uses web pages or mobile applications as the front-end and smart contracts as the back-end [3]. Yet, due to the potential inconsistent behaviors of DApps, users are exposed to the following risks.

First, the DApp front-end may generate transactions inconsistent with users’ intentions. For example, a malicious phishing DApp usually claims that users can claim some cryptocurrency for free on their websites. However, when a victim visits the front-end of the DApp and clicks the `Claim` button. Contrary to what the victim expected, the DApp not only did not send any cryptocurrency to his account but instead generated a transaction that authorized the phishing DApp to control all assets owned by the victim. As a result, the phishing DApp easily steals them.

Second, when handling the users’ transactions, the smart contract may have misbehaviors that are inconsistent with users’ intentions. For example, assuming the user publishes a transaction via the front-end to invoke the `approve` function of a DApp’s smart contract. The smart contract is expected to allow an address specified by the caller to use the caller’s cryptocurrencies (also known as tokens or coins) and sends a notification. However, if the smart contract does not perform the above behaviors, or does something else that it should not, intentionally or unintentionally, the contract is considered to have misbehaviors, which usually causes not only confusion but also severe financial losses to users.

We used the following motivating example to illustrate DApp’s inconsistent behavior. In April 2022, attackers hacked into the Discord account of BAYC (Bored Ape Yacht Club), a popular NFT¹ DApp, and spread dozens of fake URLs. These URLs point to scam DApps that were developed by the attackers. As shown in Fig. 1, they built a front-end page that closely resembled BAYC and claimed that the DApp would allow users to mint valuable BAYC NFTs for free. However, as shown in Fig. 2, when victims click the mint button prominently displayed on the front page, instead of generating an NFT mint transaction that the user might expect, the front-

[†]Zhou is now with the Ant Group.

[✉]Corresponding author.

¹Non-Fungible Token, which is a kind of uniquely identifiable asset.

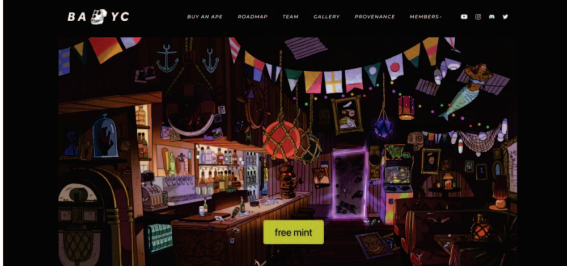


Fig. 1: The front-end of the fake BAYC scam

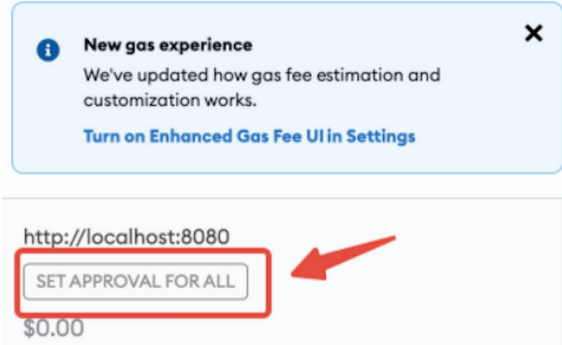


Fig. 2: The transaction generated by the scam DApp

end generates an approval transaction, which approves all the victims’ NFT to the scam DApp. This means that the scammer has full control of the victims’ NFTs and can steal the NFTs from the victims’ addresses at any time. This scam DApp had stolen six NFTs from four accounts in one day, with a total value of over \$600,000.

Real-world cases have proved that inconsistent behavior of DApps can cause severe consequences [4]–[6]. For example, in July 2022, hackers injected malicious JavaScript code into the front-end of an NFT trading DApp, PreMint, making the front-end generate malicious transactions which transfer the ownership of NFTs from the victim to the hackers. The hackers ultimately stole over 300 NFTs (worth over \$400,000).

However, no studies have been made on the inconsistent behaviors of DApps. Although there have been plenty of studies on smart contract consistency and security [7]–[12], these studies mainly focus on smart contracts, but ignore inconsistent behaviors related to the front-end of DApps and fail to examine the DApp as a whole. Such existing studies cannot detect the inconsistency between the front-end and the back-end of DApps.

This work aims to identify inconsistent behaviors of DApps by contrasting the behaviors of DApps that are derived from the DApp front-end, blockchain wallet, and smart contracts, respectively. Our approach currently focuses on the DApps that use web pages as the front-end. Identifying inconsistent behavior of DApps with mobile applications as the front-end is left as future work. First, based on user intentions (i.e. the purpose of a user for using this DApp, e.g. trading tokens),

our approach triggers the front-end to generate transactions by simulating front-end actions and infers the expected behavior in the view of user intentions. Second, our approach captures the transaction data that is sent to the blockchain wallet and extracts the behavior in the view of transaction semantics. Finally, by simulating the execution of transactions in the smart contract and analyzing transaction execution logs, our approach captures the actual behaviors of smart contracts. If any two of them do not match, inconsistent behavior is detected.

However, it is non-trivial to design and implement a practical tool to detect these inconsistent behaviors due to the following technical challenges:

C1: Complicated interaction with front-end of DApps. Due to the enormous space of possible front-end actions, it is very challenging to explore the front-end actions to correctly trigger the business logic of the DApp [13]. In addition, many features of the DApp front-end can only be used when users have sufficient cryptocurrency. However, it is at risk to use accounts that have real assets to interact with DApps with unknown security.

C2: Diversity of DApps. The business logic and usage vary greatly between DApps. In addition, still in their early stages, more types of DApps will continue to appear and evolve rapidly in the future.

C3: Consistency comparison. The behaviors retrieved from the front-end, blockchain wallet, and smart contract are implied in front-end actions, transaction data, and transaction logs respectively, which have large semantic gaps that hinder consistency comparison.

To address the above challenges, we implement our approach into a prototype named `DAppHunter` for identifying inconsistent behaviors of DApps.

To address **C1** and **C2**, we propose an *intention-driven approach* to explore feasible front-end actions for interacting with the front-end of DApps. To this end, our approach constructs a 2-layer intention-action graph, including the high-level user intention graph (UIG) and several low-level front-end action graphs (FEAG). `DAppHunter` leverages UIG and FEAG to guide the interaction with the front-end of DApps. In addition, to bypass the precondition of using the DApp front-end, we crafted a blockchain wallet to make the front-end believe that we have sufficient balance in our wallet, without putting any real assets at risk. `DAppHunter` provides a convenient way for users to easily describe the user intentions and corresponding front-end actions of a DApp in a training case. By parsing the training cases, `DAppHunter` updates the intention-action graph, thus being able to interact with DApps that have similar interaction patterns.

To address **C3**, we come up with the concept of *semantic behavior*, which has concrete types and parameters for representing behaviors of DApps. `DAppHunter` retrieves the behaviors from different sources and represents them in this unified form of semantic behavior to bridge semantic gaps.

Contributions. The main contributions of this work are as follows.

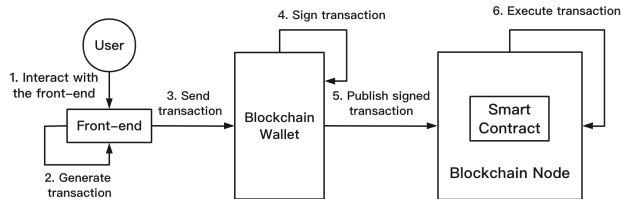


Fig. 3: The typical workflow of DApps, from a user’s perspective

- To the best of our knowledge, we take a first step to identify inconsistent behaviors of blockchain-based DApps.
- We propose a novel intention-driven approach to interact with the front-end of DApps for DApp testing.
- We implement our approach in a prototype named `DAppHunter` and evaluate it on 92 DApps on Binance Smart Chain and Ethereum. `DAppHunter` successfully identifies 37 DApps with inconsistent behaviors.

To engage the community, we released the source code of `DAppHunter` as well as the resources and experimental results at <https://github.com/HuskiesUESTC/DAppHunter>.

II. UNDERSTANDING BLOCKCHAIN-BASED DAPPS

This section briefly introduces the background knowledge of the blockchain, smart contracts, and DApps.

Account. On blockchains, an account is an entity identified by a blockchain address. There are 2 types of accounts on Ethereum [14]. (1) User accounts (controlled by private keys). (2) Contracts (deployed on blockchain and controlled by the logic of the smart contract code).

Smart Contract. A smart contract is a program that runs on the blockchain, which can provide methods to be invoked by other accounts and emit events to inform other applications that are subscribed to these events [15].

Ethereum Virtual Machine. The Ethereum Virtual Machine (EVM) is the runtime environment for EVM-compatible smart contracts [12].

Transaction. Transactions are cryptographically signed instructions from accounts [16]. It carries the following information: (1) Recipient: the receiving address. (2) Value: the amount of native token to transfer from sender to recipient. (3) Data: the data field specifies the invoked method and carries parameters when the recipient is a smart contract [16].

Blockchain Node. A blockchain node is a computer running blockchain client software. And a client is an implementation of an EVM that executes and verifies all transactions in each block [17]. All the blockchain nodes will reach the same state to reach a consensus [12].

Blockchain wallet. Blockchain wallets help users manage their blockchain accounts and sign and publish transactions to the blockchain easily. Many blockchain communities have developed their blockchain wallets (e.g. Metamask [18]).

Blockchain-based DApps. Fig. 3 shows the typical workflow of a blockchain-based DApp from a user’s perspective.

The front-end of DApp delegates its core functionality to the smart contracts deployed on the blockchain network. Users interact with the front-end to use these functions. Then, the DApp front-end generates the corresponding transactions according to the user’s actions and sends them to the user’s blockchain wallet. After the user signs the transaction using his blockchain wallet, the blockchain wallet publishes the transaction to the blockchain network. Then, the transaction will be executed in blockchain nodes.

III. RELATED WORK

To the best of our knowledge, there is no previous work focusing on inconsistent behaviors of blockchain-based DApps. Our work is partly related to automated DApp testing, smart contract verification and automated web application testing.

Automated DApp Testing. Gao et al. proposed an automated testing technique for DApps, which achieves significant optimization compared to the random testing approach [19]. Wu et al. present the first framework, `Kaya`, for testing both front-end and smart contracts at the same time. Both two methods focus on helping DApp developers to find bugs in their DApps, which test the DApps locally and require the source code of smart contracts. The two methods also pay no attention to the inconsistent behaviors of DApps. It is worth noting that their work aims to help the DApp developers to test the functionality of their code, but our work focuses on detecting inconsistent behaviors.

Smart Contract Verification. There have been a large number of approaches for smart contract verification, which can be roughly divided into 3 categories including formal verification [10], [11], [20]–[22], symbolic execution [23], and fuzzing approaches [24]–[28]. However, smart contract verification studies can only find the bugs and vulnerabilities in smart contracts, but can not identify the inconsistent behaviors in DApps.

Automated Web Application Testing. There are plenty of research results on automated testing of web applications [13], [29], [30]. Artzi et al. present `Artemis` [31], a framework that generates test inputs for JavaScript based on feedback-directed random testing. `Crawljax` is a general crawling tool for AJAX-based applications [32], which infers various paths within an AJAX application through dynamic analysis of user interface state changes. And several tools [33]–[35] were developed to test JavaScript applications based on `Crawljax`. Li et al. developed a comprehensive framework for the automatic testing of client-side JavaScript web applications [29]. These efforts can only be used to test traditional web applications, and detecting inconsistent behavior of DApps also requires bringing smart contracts and wallets into the testing scope.

IV. DESIGN AND IMPLEMENTATION OF `DAPPHUNTER`

Our approach aims to identify inconsistent behaviors of blockchain-based DApps. We implemented our approach into a prototype named `DAppHunter`. We introduce the design and implementation details in this section.

A. Definitions

For readers to better understand the following content, we first give definitions of key concepts used in DAppHunter.

User Intention: A user intention refers to the periodic purpose when using the DApp front-end.

Front-end Action: A front-end action refers to an operation on an interactive element on the front-end (e.g. click a button or input a number).

Semantic Behavior: We let τ to represent the type of behaviors of DApps (e.g. transfer, approve, etc) and let ρ to denote a parameterized description of the behavior (e.g. the value of transferred cryptocurrency, the receiver of transfer action, etc). A behavior of DApp can be denoted in $\beta(\tau, \rho)$. Two behaviors, $\beta_1(\tau_1, \rho_1)$ and $\beta_2(\tau_2, \rho_2)$ are consistent when τ_1 matches τ_2 and ρ_1 matches ρ_2 .

Front-end State & State Changes: The front-end state refers to the internal structure of elements on a front-end web page. We use the front-end state to determine whether a front-end action is feasible (e.g. whether there is an input form or a button on the web page). We further define a state change as a change on the front-end caused by either front-end action or server-side state changes propagated to the front-end.

Intention-Action Graph. Intention-action graph is used to guide the automated interaction with the front-end of DApps in DAppHunter. We define the intention-action graph as follows:

- An intention-action graph is a 2-layer graph, which consists of a high-level user intention graph (UIG) and several low-level front-end action graphs (FEAG).
- A node $N_a(state, action)$ in FEAG represents a specific front-end action. The *action* represents a front-end operation (e.g. clicking an element). And *state* represents the front-end state required to locate an element and perform the *action* (e.g. there exists an element on the page with certain keywords). An edge $E_a(N_{a1}, N_{a2})$ in FEAG denotes that, after performing front-end action N_{a1} , N_{a2} is chosen to perform.
- A node N_i in UIG represents an user intention, and an edge $E_i(N_{i1}, N_{i2})$ denotes that, after user intention N_{i1} is realized, the intention N_{i2} is chosen as the next user intention to realize.
- Each node N_i in UIG has a corresponding FEAG, and a path $(N_{a1}, N_{a2}, \dots, N_{an})$ in FEAG denotes one possible front-end action sequence to realize the user intention N_i .

B. Overview

Fig. 4 shows the architecture of DAppHunter which consists of two major parts. (1) **Front-end interaction:** Under the guidance of the intention-action graph, DAppHunter explores feasible user intention and the corresponding front-end action sequence to realize the user intention. DAppHunter uses a front-end robot to simulate the front-end action sequence and trigger the front-end to generate a transaction by simulating these front-end actions. (2) **Inconsistent behavior identification:** First, DAppHunter takes in the user intention, and corresponding front-end sequence to infer the expected behavior in the view of user intention, $\beta_e(\tau_e, \rho_e)$. Second,

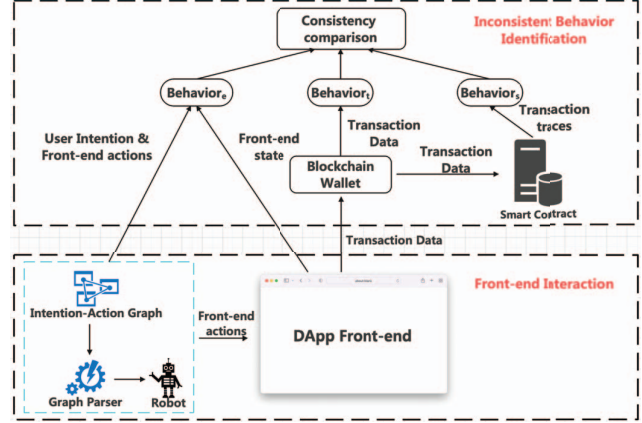


Fig. 4: Architecture of DAppHunter

by parsing the transaction data that the front-end generated, DAppHunter extracts the behavior suggested by the semantic of the transaction data, $\beta_t(\tau_t, \rho_t)$. Third, DAppHunter simulates the transaction by executing the smart contract in a blockchain node, and retrieving the transaction logs to obtain the actual behavior of the smart contract, $\beta_s(\tau_s, \rho_s)$. At last, by contrasting β_e , β_t and β_s , DAppHunter reports inconsistent behaviors if any two of them do not match.

C. Front-end interaction

To address the technical challenges C1 and C2, we designed and implemented a novel user-intention-driven approach.

Overall, our approach consists of two steps: **Step-1: Constructing and Updating the Intention-Action Graph.** DAppHunter constructs and updates the intention-action graph by parsing the DApp user intentions and interaction patterns summarized by manual analysis. **Step-2: Front-end Action Sequence Exploration.** Under the guidance of the intention-action graph, DAppHunter automatically explores the front-end operation sequence and uses a front-end robot to simulate these operations. In addition, to bypass the precondition of using the DApp front-end, we crafted a blockchain wallet to make the front-end believe that we have sufficient balance in our wallet.

The insight behind this approach is that, though the front-end action space is enormous, the optional front-end actions to realize a specific user intention is limited. Therefore, by dividing the front-end actions into different FEAGs, we can prune the huge space of front-end actions into a meaningful and feasible set of front-end action sequences. In addition, our manual analysis of about 100 DApps including exchanges, NFTs, and loans revealed the purposes of users using these DApps can be summarized by a few user intentions. The user intentions are listed in Table. I.

More importantly, the user-intention-driven approach allows us to leverage the semantic information of user intention to infer the expected behaviors of DApps. On the contrary, randomly generated front-end action sequences have very

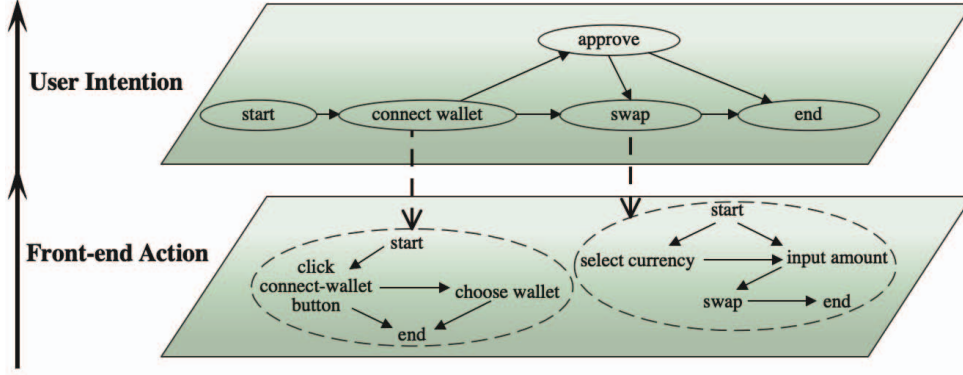


Fig. 5: The intention-action graph visualization.

little meaningful semantic information for inferring expected behaviors due to lots of ambiguous, repetitive, and pointless front-end actions.

Fig. 5 presents the simplified visualization of an intention-action graph instance. On the upper layer, DAppHunter searches for user intentions that can be realized in the UIG in the current front-end state. Then, DAppHunter explores a specific front-end action sequence that can realize the intention in the corresponding FEAG.

As shown in Fig. 4, the Graph Parser automatically explores the front-end action sequence in the intention-action graph. Then, the Robot simulates the front-end actions. Besides, the crafted MetaMask wallet enables DAppHunter to impersonate the big whale accounts that hold numerous cryptocurrencies to bypass the preconditions for using the front-end of DApps. The details of the 2 steps of our approach are as follows:

TABLE I: The list of user intentions supported by DAppHunter

Intention	Description
Connect	Connect the wallet to the front-end of this DApp
Transfer	Transfer some cryptocurrency to another address
Swap	Exchange one type of cryptocurrency for another
Approve	Approve an address to access the user's assets
Deposit	Put some assets into DApps
Withdraw	Retrieve assets from DApps
Mint	Mint cryptocurrencies or NFTs
Claim	Retrieve the profits of deposited assets

Step-1: Constructing and Updating the Intention-Action Graph. DAppHunter adopts a training process to construct and update the intention-action graph. To reduce manual efforts of constructing the graph, DAppHunter provides a convenient way for users to write training cases for DApps. A training case is a structured file similar to YAML (a markup language). Each training case consists of two parts: (1) The user intention sequence. (2) The front-end actions that realize each user intention in the intention sequence. Fig. 6 shows a simplified training case example.

```

1 intention-sequence:
2   connect
3   swap
4
5 intention:
6   name: connect
7   action-sequence:
8     click-connect:
9       operation: click
10      keywords: "connect" "wallet"
11    choose-wallet:
12      operation: click
13      keywords: "metamask"
14
15 intention:
16   name: swap
17   action-sequence:
18     select-currency:
19       operation: click
20       keywords: "currency" "select"
21     input-amount:
22       operation: input
23       keywords: "amount"
24     confirm-swap:
25       operation: click
26       keywords: "swap"
27 ...

```

Fig. 6: A simplified training case example

Each training case can be viewed as a subgraph of the intention-action graph. For example, the training case shown in Fig. 6 defines a UIG with 2 intention nodes (i.e. connect and swap, Line 2-3) and 2 corresponding FEAGs (Line 5-13, Line 15-29). DAppHunter will merge the subgraph described by the training case into the current intention-action graph so that users do not need to rewrite the training case when detecting DApps with similar interaction patterns. DAppHunter follows the following steps to merge the training case: (1) **Insert intention nodes and edges into UIG.** When the intention node defined in the intention sequence of the training case does not exist in the UIG, a new intention node will be added to the UIG, as well as its predecessor node, and edges between them will be added recursively until the predecessor node is already in the UIG. (2) **Update FEAG of intention nodes.**

The process of updating a FEAG is similar to (1), the front-end action nodes and edges denoted in the training case will be inserted into the FEAG of each intention node.

Step-2: Front-end Action Sequence Exploration. Guided by the intention-action graph and the execution result of front-end action, `DAppHunter` infers the next front-end action to realize a specific user intention. By executing this process iteratively, `DAppHunter` can explore the necessary front-end action sequence to complete a transaction.

As presented in Alg 1, starting from the start node of `UIG`, `Graph Parser` traverses `UIG` and monitors the front-end state. For each node in the `UIG`, `Graph Parser` will search for a feasible action sequence in its corresponding FEAG. First, `Graph Parser` initiates exploration from the start action node (Line 4) and then tries to find an executable element by the metadata extracted from the adjacent node (Line 7-10). Then, the element is located (Line 11), and the `Robot` will execute the corresponding action (Line 12). If it detects a front-end state change, which means the currently explored action is successfully executed, this explored action will be added to the front-end action sequence. `DAppHunter` repeats this procedure until an unrecoverable error occurs (Line 13-14), or a transaction is successfully triggered (Line 15-16). The explored front-end action sequence will be used to infer the expected behavior of the `DApp`.

Algorithm 1 Front-end Action Sequence Exploration

Input: User intention graph: `UIG`

Output: Front-end action sequence: `seq`

```

1:  $node_i \leftarrow$  start node of UIG
2: while  $node_i \neq$  end node do
3:    $fag \leftarrow$  front-end action graph of  $node_i$ 
4:    $node_a \leftarrow$  start node of  $fag$ 
5:    $seq \leftarrow \emptyset$ 
6:   while  $node_a$  is not end node of  $fag$  do
7:     for  $n \leftarrow$  adjacent nodes of  $node_a$  do
8:        $state \leftarrow$  current front-end state
9:       if  $n$  is not feasible in  $state$  then
10:        continue
11:        $element \leftarrow$  Robot.LocateElement( $n.state$ )
12:        $result \leftarrow$  Robot.Execute( $element, n.action$ )
13:       if  $result$  is unsolved error then
14:         return  $\emptyset$ 
15:       else if  $result$  is transaction triggered then
16:         return  $seq$ 
17:       else if  $result$  is state changed then
18:          $node_a = n$ 
19:          $seq \leftarrow seq + node_a$ 
20:        $state_i \leftarrow$  current front-end state
21:        $node_i \leftarrow$  NextIntentionNode( $node_i, state_i$ )

```

Bypassing the requirements in the front-end of `DApps`. Empirically, having sufficient balance to cover all expenses is the basic requirement for using `DApps`. Therefore, the front-end of a `DApp` query account balance frequently from

multiple data sources including blockchain explorers and smart contracts.

Unfortunately, using an account that holds real assets to interact with a `DApp` whose security is unknown is very dangerous. Therefore, we used a modified `MetaMask` wallet to impersonate some accounts with sufficient assets to interact with these `DApps`. In particular, the `DApps` use this function `getAccounts` to query the accounts that are managed by `MetaMask`. We first collect 2 big whale accounts that have a lot of cryptocurrencies. Then, we crafted the function and replaced the real accounts with collected big whale accounts. Therefore, when the `DApp` front-end tries to query the user's account from `MetaMask`, they will get these big whale accounts and query their cryptocurrency balance.

D. Identifying Inconsistent Behaviors

To identify inconsistent behaviors, we first extract three behaviors from three different sources: front-end, blockchain wallet, and smart contract.

The major challenge (C3) of identifying inconsistent behaviors is the large semantic gap between low-level data (i.e. the transaction data sent to the blockchain wallet and the logs of transactions retrieved from the smart contracts) and the very high-level user intentions and front-end actions retrieved from the front-end.

Bridging the Semantic Gap. To address the technical challenge C3, we come up with the concept of semantic behavior to bridge the semantic gap.

In particular, `DAppHunter` currently supports 8 types of user intentions. Except for the `connect` intention, 7 of them will generate transactions. Therefore, we defined 7 types of semantic behaviors. The mapping of user intention, transaction data, transaction log, and semantic behavior is shown in Table. II.

`DAppHunter` adopts a workflow containing 4 stages for identifying inconsistent behaviors. We will use a real-world example to illustrate the workflow (see VI-A).

Stage-1: Inferring Expected Behavior. The semantic of expected behavior is inferred from the front-end action sequence generated by `Graph Parser`. First, according to the user intention of the front-end action sequence, `DAppHunter` inferred the type of the semantic behavior. `DAppHunter` will extract the values from the front-end states and the values that `Robot` entered to infer the parameters of the semantic behavior.

For the real-world example in VI-A, `DAppHunter` identified the user intention `approve` by locating the action of clicking `Approve` button in the front-end action sequence. By extracting the token name from the front-end state, `DAppHunter` identified cryptocurrency token to be approved is `ZEPE` (shown in Fig. 8). Note that, since there is no information about the `spender` parameter on the front-end, it will be set to unknown. Therefore, `DAppHunter` inferred the expected behavior in the form of semantic behavior, i.e. `Approve(asset: ZEPE, spender: unknown)`.

TABLE II: The mapping of semantic behavior, user intention, transaction data, and transaction logs

User Intention	Transaction Data Keywords	Transaction Log Conditions	Semantic Behavior Type(Parameters)
Transfer	transfer	(Exist Transfer event E and $E.receiver = receiver$ and $E.value = value$) or (Exist Transaction Tx and $Tx.to = receiver$ and $Tx.value = value$)	Transfer($receiver, value$)
Swap	swap, exchange	Exist Transfer events sequence E_1, E_2, \dots, E_n and $E_1.asset = asset_{in}$ and $E_n.asset = asset_{out}$ and $E_1.from = E_n.to$	Swap($asset_{in}, amount_{in}, asset_{out}, amount_{out}$)
Approve	approve, approval	Exist Approval event E and $E.asset = asset$ and $E.spender = spender$	Approve($asset, spender$)
Deposit	deposit, transfer, approve	(Exist Transfer event E and $E.from = depositor$ and $E.amount = amount$ and $E.asset = asset$) or (Exist Deposit event E and $E.depositor = depositor$ and $E.asset = asset$)	Deposit($asset, amount$)
Withdraw	withdraw	Exist Transfer event or Withdraw event E and $E.receiver = receiver$ and $E.asset = asset$ and $E.amount = amount$	Withdraw($asset, amount, receiver$)
Mint	mint, transfer	Exist Transfer event E and $E.to = receiver$ and $E.from = NULL\ address$, and $E.asset = asset$ and $E.amount = amount$	Mint($asset, amount, receiver$)
Claim	claim	Exist Transfer event E and $E.asset = asset$ and $E.amount = amount$ and $E.to = DAppHunter$	Claim($asset, amount$)

Stage-2: Extracting The Behavior of Generated Transaction. The transaction data is encoded into hex bytes starting with a 4-byte method id, which is the first 4 bytes of the hash of the method’s name and parameter list. To extract transaction semantics from the transaction data, DAppHunter queries the original function name and parameter list from a method id database [36]. Then, DAppHunter uses the parameter list to reconstruct the Application Binary Interface and decodes all the parameters in the transaction data. With the method name recovered and the parameters decoded, according to the keyword in the method name and the type of parameters shown in Table. II, DAppHunter maps the transaction into a semantic behavior suggested by transaction data.

For the example in VI-A, DAppHunter recovered the function name, Transfer, and decoded the receiver (i.e. 0xBca..33AE) and value (i.e. 2871.99) from the transaction data. Therefore, DAppHunter extracted the behavior of the transaction data in the form of semantic behavior, i.e. Transfer($receiver: 0xBca..33AE, value: 2871.99$).

Stage-3: Detecting Behavior of Smart Contract. To detect smart contract behaviors, DAppHunter uses the eth_sendTransaction and eth_getTransactionReceipt remote process calls to execute the transaction on the local node and retrieve its execution logs. DAppHunter analyzes the logs to detect the events that are emitted during the transaction execution. Similarly, according to whether the transaction logs meet the conditions shown in Table. II, DAppHunter first maps the logs into a semantic behavior. And the parameters of the semantic behavior are extracted from the execution logs.

For the real-world example in VI-A, after simulating the transaction data in stage-2 in the local node and analyzing the transaction logs, DAppHunter identified a transaction Tx where $Tx.to = address$ and $Tx.value = 2871.99$ in the logs. Since the transaction log meets the condition of a Transfer semantic behavior. DAppHunter retrieved the behavior of the smart contract, Transfer($receiver: 0xBca..33AE, value: 2871.99$).

Stage-4: Detecting Inconsistency. DAppHunter compares the behaviors of the results of stage-1, stage-2, and stage-3, which are represented in the form of semantic behaviors, $\beta_e(\tau_e, \rho_e)$ and $\beta_t(\tau_t, \rho_t)$ and $\beta_s(\tau_s, \rho_s)$ from the front-end, blockchain wallet and smart contract respectively. DAppHunter reports the following 3 kinds of inconsistent behaviors.

Transaction Type Inconsistency. DAppHunter compares the semantic behavior types of $\beta_e(\tau_e, \rho_e)$ and $\beta_t(\tau_t, \rho_t)$ (i.e. τ_e and τ_t). If τ_e and τ_t mismatch, DAppHunter reports a transaction type inconsistency, indicating that the front-end generates an incorrect transaction.

Transaction Parameters Inconsistency. By comparing the parameters of $\beta_e(\tau_e, \rho_e)$ and $\beta_t(\tau_t, \rho_t)$, i.e. ρ_e and ρ_t , DAppHunter reports a transaction parameter inconsistency if any parameter in ρ_e and ρ_t mismatches.

Contract Behavior Inconsistency. By comparing β_e, β_t and β_s , a contract behavior inconsistency is reported if β_s does not match any one of β_e or β_t . It should be pointed out that due to factors such as cryptocurrency price fluctuations, exchange rate changes, etc., the amount parameters of semantic behaviors such as swap and mint are considered consistent with small errors.

For the real-world example in VI-A, in stage-1, DAppHunter inferred the $\beta_e: Approve(asset: ZEPe, spender: unknown)$. However, in stage-2, the β_t retrieved by DAppHunter is Transfer($receiver: 0xBca..33AE, value: 2871.99$). Since the type of β_e is Approve, which is inconsistent with the type of β_t . Therefore, DAppHunter reported a transaction type inconsistency.

When DAppHunter detects inconsistent behavior in a DApp, it generates a text report for the users. The report contains two parts of information. (1) The types of inconsistent behavior are shown in the report. (2) The report also shows the semantic behavior types and parameters of $\beta_e(\tau_e, \rho_e)$ and $\beta_t(\tau_t, \rho_t)$ (i.e. τ_e and τ_t) to the users so that they can better understand the functionality of the DApp under testing.

V. EVALUATION

In this section, we evaluate `DAppHunter` and present the experiment results to answer the following research questions. We also give some suggestions and for wallet providers and users at the end of this section.

- **RQ1.** Can `DAppHunter` reduce the manual effort when detecting inconsistent behavior of DApps?
- **RQ2.** What is the accuracy of `DAppHunter` in identifying inconsistent behaviors of DApps?
- **RQ3.** What are the major causes of inconsistent behaviors of DApps?

A. Experiment Setup

We deploy `DAppHunter` on a laptop, with MacOS Monterey 12.3 as the operating system and Chrome browser of version 101.0.4951.54 installed. The MetaMask v10.8.2 is crafted to impersonate the big whale accounts on BSC and Ethereum.

B. DApp Subjects

We obtain the URLs of DApp projects from the DApp browser (DAppRadar), blockchain browsers (BSCscan and Etherscan), and social media (Discord, Twitter, and Telegram channels). Since the front-end page of the DApp needs to interact with the blockchain wallet through a JavaScript library `web3.js`, we used a web crawler to check if these URLs are alive, and determine if these URLs are the front-end of DApps according to whether the website contains the `web3.js` library. Finally, we collected 92 DApps with front-end at the time of the experiment from February 2022 to July 2022.

According to the functions of these DApps, we divided the collected DApps into 4 categories, including NFT, exchange, loan, and yield farming. The number of DApps in each category and the major intentions of users when using them are listed in Table III.

TABLE III: The Major user intentions of collected DApps

Category	# of DApps	Major intentions
NFT	12	Mint, Transfer, Approve
Exchange	67	Swap, Approve
Loan	6	Stake, Withdraw, Approve
Yield Farming	7	Deposit, Claim, Approve

C. Results

1) *RQ1: Can `DAppHunter` reduce the manual effort when detecting inconsistent behavior of DApps:* To answer RQ1, we conducted a case study to compare `DAppHunter` with the existing DApp automated testing framework.

Kaya [37] is an automation testing framework for DApps. Using Kaya, users can describe the front-end behaviors and the expected value of variables in the smart contract in an XML-like file. Then, Kaya will parse and simulate the front-end behaviors in the web browser. Since the process of `DAppHunter` and Kaya’s automatic interaction with the front-end is similar, we choose to use Kaya for comparison to

evaluate whether `DAppHunter` can reduce the manual effort required for automatic interaction with the front-end.

In the case study, we have one of the authors learn how to use Kaya. Then, we sampled 10 exchange DApps from the collected 92 DApps, and let the author write and debug the testing cases for them until Kaya can automatically perform a `swap` transaction on the front-end. Then, we asked the author to write training cases using `DAppHunter` for the 10 exchange DApps as well. The simplified testing case example of Kaya is shown in Fig. 7.

As shown in Fig. 7, users of Kaya need to explicitly specify the elements on the page that need to be manipulated in a front-end action, and provide their attributes such as class, id, or tag so that Kaya can locate them on the page. To complete a `swap` transaction for one DApp, the author needs to add 10-12 front-end action items to the testing case. And each front-end action item is completed in 10 lines of XML on average. To sum up, the author needs to write about 1070 lines of XML to enable Kaya to automatically interact with the 10 DApps.

In contrast, when using `DAppHunter`, the author first writes a training case for one of the 10 DApps. In this training case, the author uses 12 lines of YAML to describe 2 intention nodes (i.e. `connect` and `swap`) and their connections. Then, the author uses 48 and 77 lines of YAML to construct the FEAG of `connect` and `swap` respectively. After this process, `DAppHunter` constructed the initial intention-action graph. Then, the author used `DAppHunter` to test the 10 DApps and found that the initial intention-action graph has been able to complete a `swap` transaction on 7 DApps. Then, the author gradually adds training cases in the same way. After adding 4 training cases, `DAppHunter` has been able to complete the `swap` transaction on all 10 DApps since the intention-action graph has covered all the front-end action paths to complete a `swap` transaction. Finally, the author used 4 training cases, 468 lines of YAML in total to enable `DAppHunter` automatically interact with the 10 DApps.

In addition, in this case study, we found that Kaya has the following disadvantages compared with `DAppHunter`. (1) Since Kaya can not bypass the balance checking in 7 of 10 DApps, it failed to complete the `swap` transaction on their front-end pages. The author has to intervene manually to make the test continue. (2) The author has to write 10 testing cases one by one when using Kaya. In contrast, since `DAppHunter` constructs the intention-action graph and is able to interact with similar DApps without writing testing cases one by one, the author only needs to complete 4 training cases for `DAppHunter`.

Therefore, it can be concluded that, compared to Kaya [37], `DAppHunter` requires almost no additional manual intervention and can bypass the balance check of most DApps, and reduces the manual effort when testing exchange DApps (10 testing cases vs 4 training cases, 1070 lines vs 468 lines).


```

1 <Transactions type="list">
2   <item type="dict">
3     <name type="str">Swap Token</name>
4     <url type="str">https://ampleswap.com/swap</url>
5     <items type="list">
6       ...
7       <item type="dict">
8         <area type="list">
9           <item type="dict">
10            <class-name type="str">swap-currency-output</class-name>
11          </item>
12        </area>
13      <tag_name type="str">input</tag_name>
14      <filter_key type="str">class</filter_key>
15      <filter_value type="str">token-amount-input</filter_value>
16      <action type="str">insert</action>
17      <action_value type="str">0.01</action_value>
18    </item>
19    <item type="dict">
20      <area type="list">
21        <item type="dict">
22          <class-name type="str">swap-page</class-name>
23        </item>
24      </area>
25      <tag_name type="str">button</tag_name>
26      <filter_key type="str">id</filter_key>
27      <filter_value type="str">swap-button</filter_value>
28      <action type="str">click</action>
29    </item>
30    ...
31  </items>
32 </item>
33 </Transactions>

```

Fig. 7: A testing case example of Kaya

Answer to RQ1. Fewer lines of code are required when writing test cases compared to the existing DApp testing framework. And DAppHunter requires almost no additional manual intervention.

2) *RQ2: What is the accuracy of DAppHunter in identifying inconsistent behaviors of DApps:* With the intention-action graph initialized, we conducted an evaluation experiment on the collected 92 DApps. Of the 92 DApps, DAppHunter failed to trigger transactions on the front-end of 33 DApps, including 7 NFTs, 19 exchanges, 4 loans, and 3 yield farming. The reasons for not triggering the transaction are as follows: (1) 3 NFT DApps require users to first register an account managed by the project party, and then associate it with their blockchain accounts. (2) The smart contracts of 14 exchanges, 2 loans and 1 yield farming DApps do not have enough funds for swap or withdraw transactions, resulting in the front-end refusing to generate transactions. (3) Due to the failure of the front-end or server, the DApp front-end did not respond to DAppHunter’s operations on the front-end.

There are 37 inconsistent DApps detected by DAppHunter, including 35 scam DApps and 2 DApps whose front-end was hacked. The results are shown in Table. IV.

We define a false negative as a DApp that behaves inconsistently but is regarded as consistent by DAppHunter. And a false positive refers to a DApp that behaves consistently but is regarded as an inconsistent one by DAppHunter mistakenly due to incorrect front-end actions, mistaken inference of expected behavior, and analysis of contract behavior. We manually checked the DApps that are reported to be inconsistent to evaluate the accuracy of DAppHunter.

As shown in Table. IV, DAppHunter successfully detected 37 inconsistent DApps. With manual analysis of the DApps that are reported to be consistent, we also confirmed that DAppHunter did not produce any false negatives. As of August 2022, due to reports from victims, the URLs and smart contracts of the 35 scam DApps have been blacklisted and

TABLE IV: Results of detected inconsistent DApps (P - Positive, N - Negative, TP – True Positive, FP – False Positive, TN – True Negative, FN - False Negative)

Category	Samples(P/N)	Reported	TP	FP	TN	FN
NFT	12(2/10)	2	2	0	10	0
Exchange	67(34/33)	34	34	0	33	0
Loan	6(6/0)	0	0	0	6	0
Yield farming	7(1/6)	1	1	0	6	0

tagged by Peckshield [38] and MetaMask [39].

Answer to RQ2. DAppHunter is accurate in identifying inconsistent behaviors of DApps.

3) *RQ3: What are the major causes of inconsistent behavior of DApps:* We further manually investigated the reported inconsistent DApps and revealed the following major reasons behind the inconsistent behaviors.

R1. Scams. Among the 37 detected inconsistent DApps in our experiments, 35 of them are scams. These scams either generate inconsistent transactions or deploy smart contracts that have misbehaviors [6], [40], [41].

R2. Front-end hacking. In our experiment, 2 inconsistent DApps are DApps whose front-end is hacked. One hacked front-end was recreated based on the malicious script disclosed by the blockchain wallet provider, ZenGo [42]. And another hacked DApp was found via the experiment result. Hackers can hack the front-end of DApps to inject malicious code into the front-end page, which can generate transactions that are inconsistent with the users’ intentions, and steal the mnemonic phrase of users’ wallets.

Answer to RQ3. Manual investigation shows that there are 2 major reasons behind the inconsistency, including scams and front-end hacking.

D. Recommendations

Recommendations for wallet providers. The blockchain wallet is currently the only line of defense for users against inconsistent transactions. Therefore, when asking users to confirm sensitive transactions, wallets should explicitly remind users of the risks of confirming the transaction. From the experimental results, no front-end (even if it contains malicious scripts) can pose a threat to the security of the wallet. However, attackers often use some social engineering methods to trick blockchain wallet users into handing over mnemonic words or private keys. In addition, since the private key is stored in the blockchain wallet, we strongly recommend that the blockchain wallet provider use a secure private key encryption storage algorithm to prevent attackers from cracking the user’s private key through the wallet’s storage file.

Recommendations for DApp users. The DApp users are suggested to always double-check the DApps that are connecting to their wallets and not confirm transactions unless they are 100% sure of what they are doing. More importantly, never

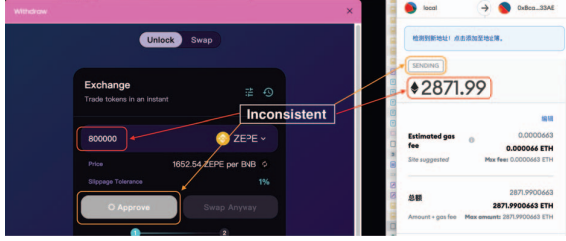


Fig. 8: The front-end of ZEPe.io

```

1 function batchTransfer(address[] memory holders
  , uint256 amount) public payable {
2   for (uint i = 0; i <holders.length; i++) {
3     emit Transfer(address(this), holders[i],
      amount)
4   }
5 }

```

Fig. 9: The batchTransfer function in the smart contract of ZEPe.io

trust DApps that are too good to be true (e.g. a huge amount of airdropped tokens).

VI. CASE STUDIES ON INCONSISTENT DAPPS

We further conducted case studies of the identified inconsistent DApps for measuring the financial losses that they caused. The partial list of these DApps is shown in Table. V. The full list can be found at <https://github.com/HuskiesUESTC/DAppHunter>.

By analyzing the related transactions with these DApps, we summarized 2 categories of scam DApps that have not yet received widespread attention, named fake airdrop and hidden fee arbitrage scam DApps.

A. Fake Airdrop Scams

An airdrop, in the cryptocurrency business, is a marketing stunt that involves sending coins or tokens to blockchain accounts to promote awareness of a new cryptocurrency. However, some phishing scams leverage the airdrop to lure victims to use their DApp front-end which contains malicious logic. The ZEPe.io listed in Table. V is an example of fake airdrop scams. It airdropped 80,000 ZEPe tokens to each victim’s account. The scammer also created a pool of

```

1 function balanceOf(address account) public view
  virtual override returns (uint256) {
2   if (!_unlocked[account]) {
3     // this is a constant number 80,000
4     return _airdropAmount;
5   } else {
6     return _balances[amount];
7   }
8 }

```

Fig. 10: The balanceOf function in the smart contract of ZEPe.io

ZEPe and BNB (the native token of Binance Smart Chain) on PancakeSwap (the most popular decentralized exchange market on Binance Smart Chain). A pool of two kinds of tokens is used to exchange one kind of token for the other. When the victims noticed that many ZEPe tokens are airdropped to their wallets, they wanted to trade them for BNB on PancakeSwap. However, their transaction of exchanging ZEPe will fail and the error message asked them to go to the front-end of ZEPe.io to approve the ZEPe token first.

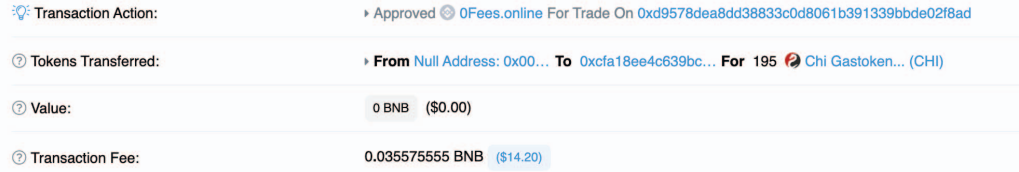
As shown in Fig. 8, when the victims use the front-end to approve the ZEPe token by clicking the eye-catching Approve button on the page, the front-end is expected to generate a transaction that approves the ZEPe token to the address of a cryptocurrency exchange smart contract. However, the malicious front-end generates a transfer transaction to send all the BNB (the native cryptocurrency of Binance Smart Chain) in this victim’s wallet to the account of the scammer. The semantic of the former is to approve an account to access the user’s ZEPe token while the semantic of the latter is to transfer BNB to the scammer.

We analyzed the transactions related to ZEPe.io and found that the smart contract of this DApp has airdropped its ZEPe token to 5,673,269 addresses using a function named batchTransfer. Under normal circumstances, the transaction fees required to make so many transactions are estimated to be more than \$60,000. However, we found the scammer only spent about \$5,100 to make all these transactions, which is quite anomalous. Since the smart contract of ZEPe.io is not open-source, we decompiled its smart contract for inspection. As shown in Fig. 9, the batchTransfer function of the smart contract does not transfer any tokens to the victim’s wallet but merely emits a Transfer event to mislead the victim’s wallet that it has received the tokens. The transaction fee required for merely emitting a Transfer event is 10 times less than executing a normal transfer transaction. Fig. 10 shows the balanceOf function of the smart contract. The function is expected to return the number of tokens held by account. However, when the victim calls the balanceOf function of ZEPe.io to query their ZEPe token balance, it checks whether the account is unlocked. Since only the accounts of the scammers are unlocked, it returns a fixed fake value.

Since the smart contracts of fake airdrop scam DApps only emit the Transfer event instead of transferring tokens, the cost of deploying a fake airdrop scam DApp and conducting a massive fake airdrop is much lower than what is generally believed. Second, since the low cost of deploying a scam DApp, the scammers tend to shut down DApps quickly after several victims have taken the bait and used the profits from the previous scam DApp to redeploy a new one to avoid being blacklisted.

B. Hidden fee arbitrage scams

The OFees listed in Table. V is an example of hidden fee arbitrage scams. The scam DApps of this category exploit inconsistencies between transaction and smart contract



Transaction Action:	Approved Fees.online For Trade On 0xd9578dea8dd38833c0d8061b391339bbde02f8ad
Tokens Transferred:	From Null Address: 0x00... To 0xcfa18ee4c639bc... For 195 Chi Gastoken... (CHI)
Value:	0 BNB (\$0.00)
Transaction Fee:	0.035575555 BNB (\$14.20)

Fig. 11: The approve transaction of 0Fees, obtained on BSCscan

TABLE V: The partial list of inconsistent DApps

DApp-Name	Front-End-URL	Contract Address	Type of Inconsistency
VeloChain	VeloChain.io	0xc7ef1bff46cd025509cf5e55fa5cd5c14793cbff	Transaction Type Inconsistency
GoFlux	GoFlux.io	0xb16600c510b0f323dee2cb212924d90e58864421	Transaction Type Inconsistency
VERA	TheVera.io	0x0df62d2cd80591798721ddc93001afe868c367ff	Transaction Type Inconsistency
EVER	TheEver.io	0x5190b01965b6e3d786706fd4a999978626c19880	Transaction Type Inconsistency
AIR	BestAir.io	0xbc6675de91e3da8eac51293ecb87c359019621cf	Transaction Parameter Inconsistency
ZEPE	Zepe.io	0x119e2ad8f0c85c6f61afd0df69693028cdc10be	Transaction Type Inconsistency
0Fees	0fees.online	0xcfa18ee4c639bc1f058a25da5dba26d8a4c895be	Smart Contract Misbehavior
0Army	0Army.io	0x4a5fad6631fd3df66f23519608185cb96e9a687d	Smart Contract Misbehavior
BNBRoyal	BNBRoyal.io	0x8bfd23fef18086c912757d91f82e903c7ce8fc6	Transaction Parameter Inconsistency

TABLE VI: The detected hidden fee arbitrage scams

DApp-Name	Address	Profit \$	Loss of Victims \$	# of Victims
0Fees	0xcfa18ee4c639bc1f058a25da5dba26d8a4c895be	2,911	4,657	302
0Army	0x4a5fad6631fd3df66f23519608185cb96e9a687d	2,043	3,268	212
CGB	0x7e6202903275772044198d07b8a536cc064f8480	12,684	20,294	1,997

behaviors. The 0Fees first sent airdrops to the victim’s address. When a victim tried to trade these tokens on an exchange market, they were asked to approve the received token to the exchange market, which is harmless under normal circumstances. However, The approve function of 0Fees’s smart contract that handles the approve transaction has malicious logic.

Fig. 11 shows an approve transaction that a victim sent to 0Fees. It is worth noting that the transaction fee is \$14.20, which is quite anomalous. Normally, an approve transaction should cost between \$0.02 and \$0.1.

Since the smart contract of 0Fees is not open-sourced, we decompiled its bytecode and found that the approve function of 0Fees fires a transaction to mint CHI tokens, which consumes a huge amount of transaction fee. And all the minted CHI tokens were sent to the contract address of the scam DApp, which is inconsistent with the user’s intention.

Other detected hidden fee arbitrage scams are shown in Table. VI, it can be concluded that the number of victims of these three scam apps exceeds 2,000. This is because the hidden fee arbitrage scams leverage the smart contract misbehaviors, which are more stealthy and harder to spot. Due to the insidious nature of this scam model, more and more scam projects like 0Fees are emerging on the Binance Smart Chain, which needs to be brought to the attention of the community.

To sum up, the inconsistent DApps, especially those scams,

have caused severe financial losses. By investigating the transactions related to the scammers’ accounts, we estimated that the 37 inconsistent DApps caused a loss of over \$1.1 million in total. Since the cost of creating and running a scam DApp is much lower than we thought, scammers can easily rebuild new scam DApps quickly after several hits to avoid being blacklisted. We will keep improving DAppHunter to help the community identify scams and other inconsistent DApps.

VII. CONCLUSION

In this work, we proposed DAppHunter, a prototype for identifying inconsistent behaviors of blockchain-based DApps. Our approach identifies inconsistent behaviors of DApps by contrasting the behaviors from the DApp front-end, blockchain wallet, and smart contract. This work has some limitations. We admit that the experiment on the ease of use of this work may introduce a bias since ”manual effort” is a subjective metric. Our study requires some manual effort to construct and update the intention-action graph and we will try to automate the whole process in our future work.

ACKNOWLEDGMENT

This work is supported by the Ant Group. And Ting Chen is partially supported by National Key R&D Program of China (2018YFB0804100), National Natural Science Foundation of China (61872057).

REFERENCES

- [1] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [2] P. Herrera, “2021 dapp industry report,” 2021.
- [3] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. Leung, “Decentralized applications: The blockchain-empowered software system,” *IEEE Access*, vol. 6, pp. 53019–53033, 2018.
- [4] SlowMist, “Over 2,101 btc were stolen from the @badgerdao,” 2021.
- [5] Badger, “Badgerdao exploit technical post mortem,” 2021.
- [6] LaserDesk, “Scam help book.”
- [7] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, “Vandal: A scalable security analysis framework for smart contracts,” *arXiv preprint arXiv:1809.03981*, 2018.
- [8] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “Zeus: analyzing safety of smart contracts,” in *Ndss*, pp. 1–12, 2018.
- [9] G. Rosu, “K: A semantic framework for programming languages and formal analysis tools,” *Dependable Software Systems Engineering*, vol. 50, pp. 186–206, 2017.
- [10] Z. Liu and J. Liu, “Formal verification of blockchain smart contract based on colored petri net models,” in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, pp. 555–560, IEEE, 2019.
- [11] T. Sun and W. Yu, “A formal verification framework for security issues of blockchain smart contracts,” *Electronics*, vol. 9, no. 2, p. 255, 2020.
- [12] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, “Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum,” in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pp. 1503–1520, 2019.
- [13] M. Billes, A. Møller, and M. Pradel, “Systematic black-box analysis of collaborative web applications,” *ACM Sigplan Notices*, vol. 52, no. 6, pp. 171–184, 2017.
- [14] ethereum.org, “Ethereum accounts,” 2022.
- [15] C. Dannen, *Introducing Ethereum and solidity*, vol. 1. Springer, 2017.
- [16] Ethereum.org, “Transactions.”
- [17] Ethereum.org, “Nodes and clients.”
- [18] MetaMask.io, “A crypto wallet & gateway to blockchain apps.”
- [19] J. Gao, H. Liu, Y. Li, C. Liu, Z. Yang, Q. Li, Z. Guan, and Z. Chen, “Towards automated testing of blockchain-based decentralized applications,” in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, (Los Alamitos, CA, USA), pp. 294–299, IEEE Computer Society, may 2019.
- [20] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, *et al.*, “Formal verification of smart contracts: Short paper,” in *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pp. 91–96, 2016.
- [21] T. Abdellatif and K.-L. Brousic, “Formal verification of smart contracts based on users and blockchain behaviors models,” in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, IEEE, 2018.
- [22] X. Li, C. Su, Y. Xiong, W. Huang, and W. Wang, “Formal verification of bnb smart contract,” in *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*, pp. 74–78, IEEE, 2019.
- [23] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254–269, 2016.
- [24] Y. Huang, B. Jiang, and W. K. Chan, “Eosfuzzer: Fuzzing eosio smart contracts for vulnerability detection,” in *12th Asia-Pacific Symposium on Internetware, Internetware’20*, (New York, NY, USA), p. 99–109, Association for Computing Machinery, 2020.
- [25] B. Jiang, Y. Liu, and W. Chan, “Contractfuzzer: Fuzzing smart contracts for vulnerability detection,” in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 259–269, IEEE, 2018.
- [26] M. Ding, P. Li, S. Li, and H. Zhang, “Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection,” in *Evaluation and Assessment in Software Engineering, EASE 2021*, (New York, NY, USA), p. 321–328, Association for Computing Machinery, 2021.
- [27] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, “sfuzz: An efficient adaptive fuzzer for solidity smart contracts,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 778–788, 2020.
- [28] Z. Wei, J. Wang, X. Shen, and Q. Luo, “Smart contract fuzzing based on taint analysis and genetic algorithms,” *Journal of Quantum Computing*, vol. 2, no. 1, p. 11, 2020.
- [29] G. Li, E. Andreassen, and I. Ghosh, “Symjs: Automatic symbolic testing of javascript web applications,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, (New York, NY, USA), p. 449–459, Association for Computing Machinery, 2014.
- [30] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for javascript,” in *2010 IEEE Symposium on Security and Privacy*, pp. 513–528, IEEE, 2010.
- [31] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, “A framework for automated testing of javascript web applications,” in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 571–580, 2011.
- [32] A. Mesbah, E. Bozdag, and A. Van Deursen, “Crawling ajax by inferring user interface state changes,” in *2008 Eighth International Conference on Web Engineering*, pp. 122–134, IEEE, 2008.
- [33] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, “Pythia: Generating test cases with oracles for javascript applications,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 610–615, IEEE, 2013.
- [34] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah, “Leveraging existing tests in automated test generation for web applications,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 67–78, 2014.
- [35] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, “Jseft: Automated javascript unit test generation,” in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, IEEE, 2015.
- [36] 4byte.directory, “Ethereum signature database,” 2022.
- [37] Z. Wu, J. Zhang, J. Gao, Y. Li, Q. Li, Z. Guan, and Z. Chen, “Kaya: A testing framework for blockchain-based decentralized applications,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 826–829, 2020.
- [38] Peckshield, “Peckshield scam alert,” 2022.
- [39] Metamask, “eth-phishing-detect,” 2022.
- [40] P. Xia, H. Wang, B. Gao, W. Su, Z. Yu, X. Luo, C. Zhang, X. Xiao, and G. Xu, “Trade or trick? detecting and characterizing scam tokens on uniswap decentralized exchange,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, dec 2021.
- [41] B. Mazorra, V. Adan, and V. Daza, “Do not rug on me: Leveraging machine learning techniques for automated scam detection,” *Mathematics*, vol. 10, no. 6, p. 949, 2022.
- [42] ZenGo, “Badgerdao malicious script, https://github.com/zengo-x/badger_dao_script_analysis.”