Stop Pulling my Rug: Exposing Rug Pull Risks in Crypto Token to Investors

Yuanhang Zhou
BNRist, Tsinghua University
Beijing, China

Jingxuan Sun
Beijing University of Posts and Telecommunications
Beijing, China

Fuchen Ma
BNRist, Tsinghua University
Beijing, China

Yuanliang Chen
BNRist, Tsinghua University
Beijing, China

Zhen Yan
BNRist, Tsinghua University
Beijing, China

Yu Jiang✉
BNRist, Tsinghua University
Beijing, China

ABSTRACT

Crypto token is a digital asset used in blockchain-based decentralized applications. Today, tokens have attracted many investors and collected a large amount of money. Unfortunately, the booming token market has simultaneously spawned numerous fraudulent schemes. Rug pull is one of the well-known scams, where fraudulent developers lure investors into seemingly profitable projects and then run off with their money, leaving the investors with worthless assets. To prevent future losses, researchers in both industry and academia have attempted to expose rug pull risks in advance. However, rug pull can manifest in various scenarios during the transfer process, posing significant challenges for effective detection.

In this paper, we first conduct an in-depth study of 201 real-world rug pull incidents for their root causes, and summarize 4 common types of rug pull risks. Then, we establish a component-configurable transfer model to locate and analyze the transfer process in token contracts. Based on the model, we generate effective oracles for the 4 rug pull risks, which can overcome the interference of diverse implementation structures. We propose Tokeer, a token verification tool that implements the transfer model and oracles with datalog technique to expose rug pull risks hidden in token contracts. We apply Tokeer on real-world tokens and compare it with state-of-the-art tools: the commercial tool GoPlus and the academic tool Pied-Piper. Tokeer achieves an average of 98.0% recall and 98.9% precision, and successfully detects 27.2% more real rug pull risks in wild production, significantly outperforming the state-of-the-art tools in terms of detection accuracy and effectiveness.

1 INTRODUCTION

Crypto token is a type of digital currency that represents an asset or offers holders certain platform-specific features. Tokens are built on top of blockchain, often utilizing smart contracts to fulfill a variety of functions. As of today, the token world is reported to be worth over $3 trillion with more than 300 million investors globally [37]. The market's continual expansion is attracting more people who see the potential for lucrative returns on their investments.

However, due to the immature and unregulated nature of the token market, numerous fraudulent schemes have emerged. Among these, rug pull has become a prevalent scamming approach and has caused significant damage to the token world. Existing study of Xia et al. [66] has proven that roughly 50% of the tokens listed on Uniswap are scam tokens, most of which have performed rug pull actions. The analysis of Cernera et al. [30] also suggest that 81.2% of 1-day tokens (comprising 60% of all tokens deployed) listed on PancakeSwap contain the rug pull scam pattern.

In rug pull schemes, fraudulent developers entice investors with seemingly lucrative projects, but then abruptly siphon off the invested funds, abandon the project and run away, leaving investors with worthless assets. A well-known rug pull incident is BNB42 [62]. The project owners deployed malicious external contracts that prevented anyone but themselves from withdrawing funds. Therefore, they block the investors from selling their assets, leaving them with valueless tokens. This causes approximately 6,000 investors to lose a total of $2.78 million.

To better understand the principles of scammers' rug pull approaches, we studied 201 rug pull events occurred from Jan. 1st, 2022 to May. 31st. 2023, which have resulted in real-world losses of more than $425 million. Through our analysis, we summarize the most common root causes embedded in the token contracts that disrupt the normal transfer execution and lead to rug pulls: **BlackList**: The token records the users' transfer permissions. Users can be restricted from transferring their assets, e.g., reselling their tokens. Therefore, the users' tokens become valueless. **ModifyBalance**: Administrators can modify the balance of certain addresses. They can manipulate the users' assets without approval, or drain out the token liquidity, resulting in significant fluctuations in the token value. **TimeLimit**: By imposing specific time constraints on the transfer, developers prevent users from transferring their assets, thereby rendering them worthless. **AlienDepend**: The transfer process invokes an external contract that isn't publicly available and whose address can be assigned by scammers. Therefore, scammers can enable malicious functionalities out of sight.

Exposing rug pull risks is critical to safeguarding investors' assets, and both academia and industry have gained some related achievements. Xia et al. [65] and Cernera et al. [30] identifies rug pull tokens through machine learning and pattern matching based on the previous transactions. Their approaches are limited to identifying rug pull behaviors that have already occurred. There are also existing tools that can expose scamming risks implanted in the token contract. Pied-Piper [51], for instance, applies datalog analysis based on the contract's EVM bytecode to reveal backdoor threats, some of which are equivalent to rug pull risks. However, its plain detection strategy cannot accurately identify the transfer execution flow, and it lacks effective oracles. As a result, it is unable to handle diverse and complex real-world scenarios. Besides, the industry

✉Yu Jiang is the corresponding author.

also offers commercial solutions for auditing smart contracts to identify potential scamming risks. Tools like GoPlus Security [57], TokenSniffer [48], RugPullDetector [1] audit the source code of token contracts to check token security. Some of them can cover the check of several rug pull risks. However, heavily depending on the contract source code and known rug pull patterns, they are susceptible to source-code level obfuscation and cannot cover complex code structures. In conclusion, existing tools are not effective enough in accommodating various rug pull scenarios, resulting in lots of false negatives and false positives.

To effectively detect rug pull risks in token contracts, there are two main challenges. **First, the modeling of a generalized token transfer process is challenging.** Transfer is the core process of token transfer, where rug pulls always occur. Therefore, locating and analyzing the transfer in token contracts is essential for subsequent detection. However, the structure of transfer implementation varies in practice due to the existence of multiple sub-processes with uncertain sequential relationships. **Second, it is challenging to define effective oracles that can expose rug pull risks that are concealed within diverse implementations.** Scammers nowadays frequently employ unconventional code implementations or obfuscation techniques to conceal fraudulent code snippets, which poses great challenges in devising an oracle that can encompass all possible code structures. For example, to restrict an address from transferring, fraudulent developers may block it at any point before or during the transfer process. This can include restricting the address from entering the transfer core process or terminating the execution before the balance update.

To address the challenges, we propose a component-configurable transfer model that is applicable across various scenarios. It identifies the code components that implement transfer functionality and establishes their execution flows. Based on the transfer model, we propose an oracle generation strategy, which generates oracles that accommodate various code structures. We apply our model and oracle generation strategy to the 4 rug pull risks and propose Tokeer, which automatically exposes rug pull risks in crypto tokens. Specifically, Tokeer can be applied to all the solidity-based [39] contracts by performing datalog analysis on the intermediate representation (IR) generated from the contract's Ethereum Virtual Machine (EVM) bytecode.

For evaluation, we apply Tokeer on real-world tokens from Binance Smart Chain (BSC) [32] (the most active chain [30] with over 3 million transactions and 700 newly verified contracts per day), and tokens that have caused rug pull events. We evaluate Tokeer's detection accuracy and detection effectiveness, and compare it with the commercial tool GoPlus and the academic tool Pied-Piper. Tokeer achieves a 98.0% recall, 44.9% better than GoPlus and 91.6% better than Pied-Piper, and a 98.9% precision, 0.6% better than GoPlus and 52.3% better than Pied-Piper. Furthermore, Tokeer can expose all the rug pull risks in tokens that have led to real-world rug pull events, and can identify 27.2% more rug pull risks than GoPlus and Pied-Piper in real-world deployed tokens.

In summary, this paper makes the following contributions:

- We conduct in-depth studies on 201 rug pull events and summarize 4 common types of rug pull risks.
- We propose a component-configurable transfer model and oracle generation strategy for rug pull detection. Based on them, we implement Tokeer, which utilizes datalog analysis to expose rug pull risks in tokens.
- We evaluate Tokeer on real-world tokens and compare it with state-of-the-art tools. Tokeer achieves better accuracy and finds more real rug pull risks. We have open-sourced Tokeer[1] and our datasets.

2 STUDY ON REAL-WORLD RUG PULLS

Previous studies on rug pull have predominantly focused on the phenomena and consequences of rug pulls. There is a lack of systematic study of the root causes behind real-world rug pull events, specifically the scamming techniques implanted in token contracts. Therefore, we conduct an in-depth study on real-world rug pull events that have already caused huge losses. The goal is to delve into the fundamental rug pull causes in token contracts to guide us in exposing rug pull potential before it leads to actual loss.

2.1 Study Methodology

In this section, we present our study methodology including the data collection and the analysis approach.

Data Collection. Our study covers 201 real-world rug pull incidents that occurred from Jan. 1st, 2022 to May. 31st, 2023, which have resulted in a total loss of over $425 million. We obtain the incidents from the analysis reports and technical posts of blockchain security companies including PeckShield [54], Beosin [6], Certik [31], Blocksec [9], Chainalysis [33] and Solidus Labs [58]. These blockchain security teams monitor the token market in real-time to provide timely alerts on the plunge of token value, which is a typical phenomenon of a rug pull. From their alerts, we obtain the basic information (e.g. contract address) of the relevant token and perform the empirical study.

Analysis Approach For each rug pull incident, we inspect their behaviors and contract source codes to study their rug pull features. Additionally, we search for existing information from blockchain security teams to help summarize the causes of rug pull. When studying a rug pull incident, we first identify whether it is caused by malicious functionalities embedded in its contract's transfer implementations. We then apply a more detailed inspection of its technical root causes. For example, how the developers block the users from selling (e.g. the branching conditions that lead to the termination of the transfer process).

2.2 Common Rug Pull Risks

Through our in-depth study, we analyze the detailed scamming approaches that lead to rug pull events. We focus on the 201 real-world rug pull events (A rug pull event can be triggered by a combination of multiple risks) that can be detected through the token contract, and summarize 4 most common types of rug pull risks:

BlackList. Some tokens allow the administrators to restrict the transfer permissions of certain addresses. They usually establish a map that can be set only by the administrators. During the transfer, the mapped value is checked and controls the branching condition.

[1]Tokeer is available at: https://github.com/TokenSecure/Tokeer

As a result, some addresses cannot transfer the assets. In our survey, 118 (58.7%) rug pull tokens contain this risk. One example is the Sirius_Finance [42], making away with more than $44,943 in customer funds. At the time of token claim, the developers blocked all addresses, not allowing them to transfer tokens. Furthermore, developers can silently sell out tokens and drain market liquidity. Therefore, the assets of users became valueless.

ModifyBalance. Administrators can arbitrarily modify the balance of a certain address without any legitimate reason or backing. Such modifications can have significant consequences. For example, they can burn a large number of tokens to drain out the token liquidity. This often triggers a swift and severe collapse in the token's price. Besides, some can even transfer away the investors' tokens to their addresses without authorization. 85 (42.3%) incidents are caused by this risk. JST [2] is an example that the scammer withdraws the user assets and transfers to the scammer's address, causing a loss of $1,150,000. Besides, tokens like NOVA [7] contain a mint function that can only be called by the owner. The owner minted a huge amount of NOVA and sold them immediately, stealing $105,811. This allows the owner to make illegal profits and causes the token value to dump.

TimeLimit. In some tokens, administrators can block the transfer by checking the timestamp of the current block. Before transferring the assets, the timestamp is obtained and is compared with the limit value after a series of calculations. The comparison result controls the branching condition and subsequently affects the transfer execution. 16 (8.0%) incidents are related to this risk. StarFinancial [56] is a typical example of rug pull caused by TimeLimit. The developer secretly changed the code to set timing limits on the transfer. The token blocks all transfers that occur after a certain time limit. This blocks any further transfers, resulting in a total loss of $40,264. Since the tokens cannot be sold, they become valueless.

AlienDepend. Some tokens implement the transfer process in an external contract or call the external functions during the transfer process. These alien contracts are often close-sourced, and their addresses can be set by the token creators through the constructor or be modified by administrators. The external functions can conduct various malicious operations, including 'BlackList', 'ModifyBalance', and 'TimeLimit'. 50 (24.9%) investigated tokens contain this type of risk. A typical case is a token CirculateBUSD [10]. It calls an unverified SwapHelper contract, in which the specified token (and amount) is transferred to a hardcoded address, causing a $2 million loss. Besides, Sirius_Finance [42] with BlackList mentioned above applies the proxy mechanism to hide the BlackList in an external, unverified contract. Therefore, the owner can constantly create rug pulls out of sight.

3 MOTIVATING EXAMPLES

Today, the serious dangers of rug pull have garnered increasing attention. Developers in both industry and academia have made attempts to identify rug pulls in tokens. Unfortunately, despite efforts, no feasible solution has been found to fully address the problem. Existing works face substantial challenges in identifying hidden rug pulls across various code structures. In this section, we highlight this problem using real-world tokens with rug pull risks.

3.1 Real-world Hidden Rug Pull Risks

Scammers nowadays always apply various unconventional code implementations to hide the scamming logic and circumvent the checks of existing works. In Fig. 1, we show the transfer execution flow of two real-world rug pull tokens. Parallelogram components represent the checking of branching conditions. Rectangular components represent function calls. Dotted components stand for optional processes controlled by branching conditions. Sometimes, the scammers check the branching conditions before entering the internal transfer function. For example, in token Dialectic [27] in Fig. 1(a), the check of mapping `_isExcluded` and `_anitswaplist` happen before entering the inner transfer process `_transfer`. Besides, scammers hide the branching conditions in recursive function calls. In TMK [60] in Fig. 1(b), the check of `sender` is wrapped in function `getCheckRoot`.

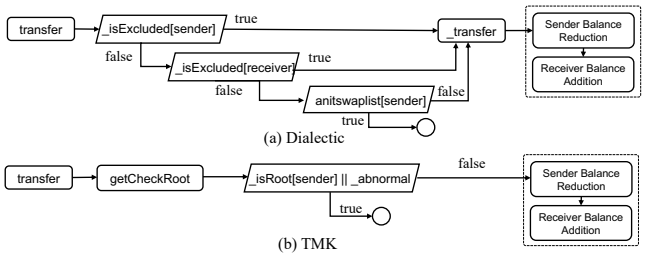

Figure 1: The transfer execution flows of real-world BlackList tokens. They cannot be detected by existing works.

3.2 Challenges to Expose Hidden Risks

Existing works like Pied-Piper and GoPlus cannot detect the rug pull risks in Fig. 1. Pied-Piper applies datalog analysis to identify backdoor threats. However, it cannot detect the rug pull risks in Fig. 1. It lacks an effective analysis of the transfer process, and cannot identify which parts of the code can affect the execution of the transfer. For example, in Fig. 1(a), the checking of BlackList mapping happens before entering the core transfer process `_transfer` and thus is falsely ignored. For Fig. 1(b), the checking of the BlackList is wrapped in other functions, which are not recognized as a part of the transfer. GoPlus performs contract auditing at source code level, matching known patterns of the rug pull risks. It can be easily confused by source code level obfuscation such as the unconventional mapping name in Fig. 1. Besides, lacking the analysis of transfer execution flow, it cannot identify the complex function invocation structures, and cannot determine the impact of certain operations on the execution flow.

In comparison, Tokeer proposes a component-configurable transfer model that facilitates the analysis of the transfer process. Firstly, the transfer model identifies commonalities across different implementations to establish a transfer framework. As depicted in Fig 1, despite variations in code structures, all implementations contain an interface that takes in sender, receiver, and transfer amount. Additionally, they all result in asset transfers, represented as `Sender Balance Reduction` and `Receiver Balance Addition`. Secondly, Tokeer identifies various components within the transfer framework, such as function calls and branching conditions. By applying control flow analysis, Tokeer identifies their specific impact on the

transfer. Tokeer then establishes an oracle generation strategy. It focuses on the functionality of each component and how it affects the transfer execution. For example, in Fig 1(b), Tokeer first identifies the function `getCheckRoot` as a part of the transfer process that can impact the execution. Then Tokeer identifies that the branching condition within it checks the sender's address and may inhibit the transfer, leading to BlackList. In conclusion, Tokeer's transfer model and oracle generation strategy overcome the existing challenges. It is effective in various rug pull scenarios, exhibiting high resilience to code obfuscation and unconventional code implementations.

4 TOKEER DESIGN

In this section, we introduce the overall design of Tokeer. As shown in Fig. 2, we first present the details of the construction of the component-configurable transfer model. Then we show the oracle generation strategy based on the model. Finally, we introduce the verification process of targeted tokens using generated oracles.

4.1 Transfer Model Construction

The transfer model is constructed in a bottom-up fashion. Modeling the transfer process allows for accurately locating and identifying the various components of transfer, which serves as the foundation for subsequent oracle definition.

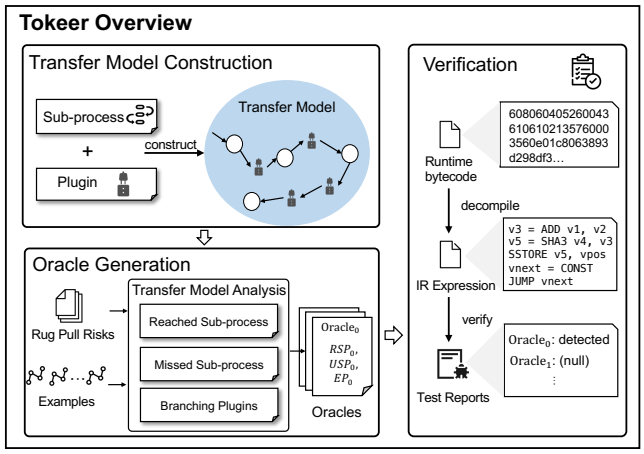

Figure 2: An overview of Tokeer. (1) Tokeer constructs the component-configurable transfer model (2) Based on the transfer model, Tokeer generates oracles for rug pull risks. (3) Tokeer conducts verification on the IR expression decompiled from the EVM bytecodes of the token contracts.

4.1.1 ***Notations.*** We begin by defining essential notations, which are the bottom-level elements that represent simple statements, operations, or relations. They include:

- *Comp*$(param_1, param_2)$: Comparing the value of $param_1$ and $param_2$. *Comp* includes *EQ*, judging if two parameters are equal; *GE*, judging if the first parameter is greater than or equal to the second one; *LE*, judging if the first parameter is less than or equal to the second one. All value relations can be covered by these three operators.

$$Comp := EQ \mid LE \mid GE$$

- *Calc*$(param_1, param_2)$: Taking $param_1$ and $param_2$ as inputs for mathematical calculations. *Calc* includes *Add*, *Sub*, *Mul*, *Div*.

$$Calc := Add \mid Sub \mid Mul \mid Div$$

- *Type*(var): Variable var's type is: *Addr*, a right-most 160 bits value that represents the address of a smart contract, a wallet, or a transaction hash; *Const*, a variable with constant value; *Global*, a global variable.

$$Type := Addr \mid Const \mid Global$$

- *Input*$(var, func)$: Variable var is a parameter of the function $func$.
- *DataFlow*(var_1, var_2): Variable var_2 is the value obtained from var_1 after a series of calculations or data flows.
- *BlockEdge*$(block_1, block_2)$: There exists an execution path that jumps from the end of $block_1$ to the beginning of $block_2$.
- *Included*(A, B): A is a smaller element than B and is included by B. This relationship can hold between a statement and a block, a block and a function, a block and a component (a sub-process or a plugin), etc.

4.1.2 ***Sub-processes.*** While the implementations of transfer processes in various tokens may differ in their internal details, they share a similar framework and functionality. For example, tokens that comply with the ERC-20 standard need to implement the transfer interfaces and events, which define the parameters, return values, and core functionalities. Therefore, we extract 4 key sub-processes (abbreviated as *SP*) that are common to all transfer implementations of tested tokens. Details are shown in Fig. 3.

- SP_0 : *CallPublicTransfer*(f): Function f is the public entry of the transfer process. Tokens that conform to ERC-20 token standard are all required to provide public transfer interfaces for users to trade their tokens. Therefore, calling the public transfer method is the first necessary step for a transfer. The method contains three parameters: address variables *sender* and *receiver* that represent the sender and receiver account addresses, and an unsigned integer *amount* denoting the transfer amount.
- SP_1 : *EnterInternalTransfer*(f_1, f_2): f_1 is the public transfer interface, and f_2 is the inner transfer method that implements the core functionalities. For a transfer process, after calling the public transfer interface, it will jump to the internal transfer method for subsequent execution.
- SP_2 : *SenderBalanceReduction*(f): Function f contains an operation that reduces the sender's balance. The sender address and amount can be identified by *Input(sender, f)* and *Input(amount, f)*. *Global(balance)* represents the global variable that records the balance of users' wallet addresses. During the transfer process, a certain amount of tokens in the sender's account will be transferred to the receiver's account. Therefore, the transfer process must have a sub-process that reduces the sender's balance. *Sub*$(balance[sender], val)$ denotes that the balance of *sender* is reduced by *val*, and *DataFlow*$(amount, val)$ denotes that the reduced value is related to the input amount.
- SP_3 : *ReceiverBalanceAddition*(f): Function f contains an operation that increases the receiver's balance. Similar to the previous, *Input*$(receiver, f)$ represents the input receiver address, and *Global(balance[receiver])* represents the global variable that records the balance of *receiver*'s wallet address. Correspondingly,

Sub-processes:

$SP_0 := Input(sender, f) \wedge Addr(sender) \wedge Input(receiver, f) \wedge Addr(receiver) \wedge Input(amount, f) \wedge Uint(amount)$

$SP_1 := Include(b_1, f_1) \wedge Include(b_2, f_2) \wedge BlockEdge(b_1, b_2)$

$SP_2 := Input(sender, f) \wedge Addr(sender) \wedge Input(amount, f) \wedge Uint(amount) \wedge Global(balance) \wedge DataFlow(amount, val) \wedge Sub(balance[sender], val)$

$SP_3 := Input(receiver, f) \wedge Addr(receiver) \wedge Input(amount, f) \wedge Uint(amount) \wedge Global(balance) \wedge DataFlow(amount, val) \wedge Add(balance[receiver], val)$

Plugins:

$P_0 := isAddrValid(addr) \mid isAddrPermitted(addr)$

$isAddrValid(addr) := EQ(addr, val) \wedge (Const(val) \vee Global(val))$

$isAddrPermitted(addr) := EQ(map[addr], val) \wedge Global(map) \wedge (Const(val) \vee Global(val))$

$P_1 := DataFlow(time, inter) \wedge LE(inter, val) \wedge (Const(val) \vee Global(val))$

$P_2 := Include(b_1, f_1) \wedge Include(b_2, f_2) \wedge BlockEdge(b_1, b_2)$

$P_3 := ExternalCall(stmt)$

$P_4 := OnlyOwner(addr) \mid isAddrPermitted(addr)$

Figure 3: Bottom-up definitions of 4 sub-processes and 5 plugins. They are defined based on the notations.

the transfer increases the buyer's balance, which is represented by $Add(balance[receiver], val)$.

4.1.3 ***Plugins.*** we define a plugin as an operation with a specific functionality. Some operations may occur multiple times at different locations during the transfer. This results in a wide variety of code implementations. To adapt to diverse transfer implementations, we introduce 5 types of plugins that enrich the transfer model. Detailed instructions are shown in Fig. 3.

- P_0 : $AddrCheck(addr)$: The check of the validity and permissions of a certain address *addr*. Let $isAddrValid(addr)$ be the validity check of *addr*. It verifies that *addr* is not an illegitimate address such as the zero address $0x0$, or a defined global variable. Let $isAddrPermitted(addr)$ be the execution permission check of *addr*. A rug pull token usually establishes a data structure to mark the privilege of addresses. In this condition, it verifies the mapped value of *addr*, and determine the branching conditions. *isAddrValid* and *isAddrPermitted* are collectively referred to as *AddrCheck*.
- P_1 : $TimingCheck(time)$: The check of a timestamp. The checking first obtains the timestamp of the current block and then performs some calculations. The result is compared with the limit, which can be a constant or a variable.
- P_2 : $InternalCall(f_1, f_2)$ Function f_1 calls function f_2, which is another function within the same token contract. To achieve $InternalCall(f_1, f_2)$, there should be b_1 in f_1 and b_2 in f_2. Besides, b_1 and b_2 satisfy the condition $BlockEdge(b_1, b_2)$.
- P_3 : $ExternalCall(stmt)$: Statement *stmt* calls a function in an external contract, which is usually not open source, and obtains the return value for subsequent execution.
- P_4 : $BreakIn(addr)$: Address with excessive privileges can directly access the transfer process to modify the balance of some users. The checking of privileges can be conducted using relation $OnlyOwner(addr)$ or $isAddrPermitted(addr)$.

4.1.4 ***Model Construction.*** The sub-processes identify the fixed components of the transfer, and the plugins extend the diversity of detailed implementations. Their combination forms the component-configurable transfer model. The 4 sub-processes exist in a certain order, while the plugins can be arbitrarily plugged into any location in the transfer process. The detailed definition used in this session is shown in Fig. 4.

The construction of the transfer model involves two steps. First, we use sub-processes to establish an underlying framework. We identify the functions where the sub-processes are located, and then establish the transfer framework. The public transfer entry is identified as $CallPublicTransfer(f_{pub})$. The inner transfer function is identified as $EnterInternalTransfer(f_{pub}, f_{in})$, which specifies the invocation hierarchy between f_{pub} and f_{in}. The functions that conduct balance modification are recognized as $SenderBalanceReduction(f_{red})$ and $ReceiverBalanceAddition(f_{add})$. In addition, we define $FunctionReach(f_1, f_2)$ to indicate that f_1 can reach f_2 through a series of function calls. Therefore, the underlying transfer framework can be presented as $Transfer(f_{pub}, f_{red}, f_{add})$, which uses f_{pub} to locate the entry point, and use f_{red} and f_{add} to ensure the transfer functionality. Besides, all functions that can be reached and executed during the transfer are identified as $FunctionReach(f_{pub}, f)$. Second, we plug the plugins into the transfer framework to cover various code structures. Let $\mathbb{F}$ be the set of functions that may be executed during transfer. As presented above, all $f \in \mathbb{F}$ can be identified by $FunctionReach(f_{pub}, f)$. Let $\mathbb{P}$ be the set of plugins recognized within the transfer process. For every $p \in \mathbb{P}$, the relation between it and its located function f_i is denoted by $Included(p, f_i)$. Up until now, the complete transfer model is constructed as $TransferModel(\mathbb{F}, \mathbb{P})$. The types of plugins can be enriched to accommodate new rug pull risks in the future.

4.2 Oracle Generation

We generate the oracle for each risk from a number of corresponding samples. For each sample, we first identify the components of its transfer process using the transfer model. Then, we use a triplet (RSP, MSP, BP) to record the analysis result, where *RSP* (Reached Sub-Processes) denotes sub-processes that can definitely be executed; *MSP* (Missed Sub-Processes) indicates the sub-processes that may not be executed due to the branching condition; *BP* (Branching Plugin) denotes the plugin that determines the branching condition. The triplet of the *j-th* sample of *i-th* risk would be $(RSP_{ij}, MSP_{ij}, BP_i)$. For the example in Fig. 1(a), the branching conditions determined by *AddrCheck* happens before *EnterInternalTransfer*. Therefore, the *RSP* is *CallPublicTransfer*, *MSP* is *EnterInternalTransfer*, *SenderBalanceReduction* and *ReceiverBalanceAddition*, and *BP* is *AddrCheck*.

Then, we calculate the intersaction of RSP_{ij} to form RSP_i, which denotes that in all senarios with *i-th* risk, RSP_i can be definitely executed. Similarly, we calculate the intersaction of MSP_{ij} to form MSP_i, which represents that once the *i-th* risk works, MSP_i must not be executed, regardless of the specific code implementations. Therefore, the oracle of the *i-th* risk should be (RSP_i, MSP_i, BP_i).

Transfer Model:

$$FunctionReach(f_1, f_2). \coloneqq (f_1 = f_2) \mid InternalCall(f_1, f_2) \mid FunctionReach(f_1, f_3) \wedge FunctionReach(f_3, f_2)$$

$$Transfer(f_{pub}, f_{red}, f_{add}) \coloneqq CallPublicTransfer(f_{pub}) \wedge EnterInternalTransfer(f_{pub}, f_{in}) \wedge SenderBalanceReduction(f_{red}) \wedge ReceiverBalanceAddition(f_{add}) \wedge FunctionReach(f_{in}, f_{red}) \wedge FunctionReach(f_{in}, f_{add})$$

$$TransferModel(\mathbb{F}, \mathbb{P}) \begin{cases} \mathbb{F} \coloneqq \{ f_i \mid Transfer(f_{pub}, f_{red}, f_{add}) \wedge FunctionReach(f_{pub}, f_i) \} \\ \mathbb{P} \coloneqq \{ p_i \mid Include(p_i, f_i) \wedge f_i \in \mathbb{F} \} \end{cases}$$

Oracle:

$$Split(b_0, b_1, b_2) \coloneqq BlockEdge(b_0, b_1) \wedge BlockEdge(b_0, b_2)$$

$$BlockReach(b_1, b_2) \coloneqq EQ(b_1, b_2) \mid BlockEdge(b_1, b_2) \mid BlockReach(b_1, b_3) \wedge BlockReach(b_3, b_2)$$

$$Oracle \coloneqq TransferModel(\mathbb{F}, \mathbb{P}) \wedge BP \in \mathbb{P} \wedge (Include(RSP, f_r) \wedge f_r \in \mathbb{F}) \wedge (Include(MSP, f_m) \wedge f_m \in \mathbb{F}) \wedge (Include(b_0, BP) \wedge Include(b_m, MSP) \wedge Split(b_0, b_1, b_2) \wedge BlockReach(b_1, b_m) \wedge \neg BlockReach(b_2, b_m))$$

Figure 4: Definition of the component-configurable transfer model and oracles. They are defined based on the notations, sub-processes, and plugins.

Once the triplet is set, we apply the following pattern to build the oracle: The transfer function executes to both *RSP* and *BP*. *BP* determines the branching condition and splits the execution into two paths. One path can execute to *MSP*, while the other cannot.

To formally define the oracles, we propose the following notations (detailed definitions in Fig. 4) to express the relations among blocks. *Split*(b_0, b_1, b_2) indicates that block b_0 jumps to b_1 and b_2 as two execution paths, respectively; *BlockReach*(b_p, b_q) denotes that there exists a path for b_p to execute to b_q. The definition of the oracle can be established as shown in Fig. 4. First, we identify the transfer model and check that *RSP*, *MSP*, *BP* are all identified within the transfer. Then we check that *BP* split the transfer flow into two, and one of them cannot reach the blocks within *MSP*. For 4 rug pull risks, we build the triplets for them as follows and apply the oracle generation strategy.

$$BlackList \coloneqq (SP_0, SP_3, P_0) \quad ModifyBalance \coloneqq (SP_{2/3}, SP_0, P_4)$$

$$TimeLimit \coloneqq (SP_0, SP_3, P_1) \quad AlienDepend \coloneqq (SP_0, SP_3, P_3)$$

For BlackList, AlienDepend and TimeLimit, the *RSP* is *CallPublicTransfer* and *MSP* is *ReceiverBalanceAddition*. They treat *AddrCheck*, *ExternalCall*, and *TimingCheck* as *BP* respectively. For ModifyBalance, its *RSP* is *SenderBalanceReduction* or *ReceiverBalanceAddition*, *MSP* is *CallPublicTransfer*, and *BP* is *BreakIn*. These oracles precisely express the fundamental logic of rug pull risks and are capable of encompassing complex code implementations.

4.3 Verification

This section illustrates the process of token verification. For each token contract, Tokeer gets the EVM runtime bytecode based on its address. Then Tokeer decompiles the bytecode into high-level IR and conducts CFG (Control Flow Graph) analysis to extract $facts$. Based on the $facts$, Tokeer starts the bottom-up analysis. Tokeer first identifies all the notations *notas*, sub-processes *SP*, and plugins *P*, in turn. Since identifying each *SP* or *P* is independent from each other, Tokeer speeds up the process by performing each identification asynchronously. After that, Tokeer construct the transfer model to locate and analyze the transfer flows. Finally, Tokeer tries to match each oracle within the transfer model. If a matchable part is found, Tokeer records the results.

Fig. 5 shows an example of how Tokeer applies datalog to IR expression for matching a defined pattern. *Input* establishes the relation that *var* is an input parameter of $func$. In this example,

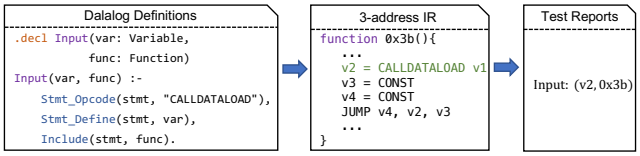

Figure 5: An example of applying datalog to IR.

the statement in green matches the definition that the statement's opcode is CALLDATALOAD. Therefore, Tokeer finds that variable *v3* and function *func* at position *0x3b* match the relation *Input(v3, 0x3b)* and outputs the results. Similarly, Tokeer defines the transfer model and oracles to match the target rug pull risks.

4.4 Implementation

This section details the implementation of Tokeer. Gigahorse [44] is a reverse compiler that decompiles smart contracts from Ethereum Virtual Machine (EVM) bytecode into a high-level 3-address IR. Tokeer first employs gigahorse to generate the 3-address IR. Then, Tokeer performs the bottom-up datalog analysis on IR and constructs the transfer model. Tokeer starts by defining the basic terms, e.g., the EVM opcode, parameters, and variables of a statement, that can be directly extracted from IR. Based on these terms, Tokeer defines the essential notations, sub-processes, and plugins, and combines them to construct the transfer model.

```
1  .decl AddrCheck(func: Function, bj: Block)
2  AddrCheck(func, bj) :-
3      isAddrValid(func, bj).
4  AddrCheck(func, bj) :-
5      isAddrPermitted(func, bj).
6
7  .decl BlackList(funcPub: Function)
8  .output BlackList
9  BlackList(funcPub) :-
10     Transfer(funcPub, funcRed, funcAdd),
11     AddrCheck(funcCheck, b0),
12     Split(b0, b1, b2),
13     Include(bm, funcAdd),
14     BlockReach(b1, bm), !BlockReach(b2, bm).
```

Figure 6: The datalog segment of BlackList oracle definition.

Next, Tokeer implements the oracles for rug pull risks with datalog. Both the transfer model and oracles are declared by souffle [46], an advanced datalog programming language. For example, Fig. 6 shows the portion of the bottom-up datalog definition of BlackList oracle. Plugin *AddrCheck* contains two conditions *isAddrValid* and *isAddrPermitted* which are defined based on the notations. Tokeer integrates the transfer model *Transfer* and plugin *AddrCheck* and establishes the oracle for BlackList based on the oracle generation strategy. Finally, Tokeer automatically applies the oracles on the transfer model and records the matched parts in the report.

5 EVALUATION

Dataset. To fully evaluate Tokeer, we provide three datasets. Our first dataset, dubbed **the survey dataset**, are 201 rug pull tokens from our study in Section 2. Our second dataset, dubbed **the full dataset**, initially consists of 10,000 newest deployed tokens from BSC up to October 14, 2022. However, since we need to manually verify the results, and GoPlus only supports open-sourced contracts, we only retain all the open-sourced ones, which are 3,562 in total. Fig. 7 shows the profile of the full dataset, where the x-axis represents the Solidity compiler version, and the y-axis represents the size of runtime bytecode. The result shows that our dataset covers compiler versions from v0.3.1 (released in Apr. 2016) to v0.8.17 (released in Sept. 2022), and bytecode size from 45B to 24,555B, demonstrating the completeness of our dataset. Our third dataset, dubbed **the sampling dataset**, consists of 234 smart contracts sampled from the full dataset. To avoid bias, we sample all tokens that the last digit of its address is 'f', and manually label the samples with their rug pull risks. Based on the sampling dataset, we conduct experiment 5.1 to evaluate Tokeer's detection accuracy. Based on the survey dataset and the full dataset, we conduct experiment 5.2 to evaluate Tokeer's detection effectiveness. Based on the full dataset, we also conduct experiment 5.3 to evaluate Tokeer's time overhead.

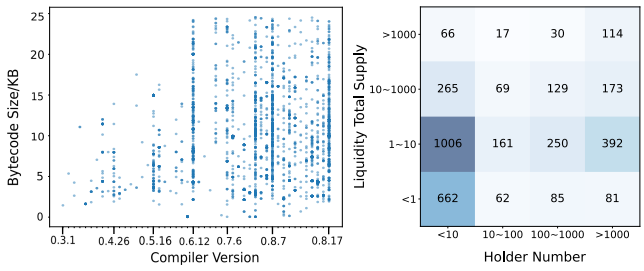

Figure 7: The profile of our evaluation dataset.This shows that we cover a wide range of contract properties.

Comparison Objects We compare Tokeer with the state-of-the-art tools that can detect rug pull risks in token contracts: GoPlus from industry and Pied-Piper from academia. GoPlus is a well-known commercial tool that provides APIs for querying the basic information and potential defects in the contract source code. Its covers 4 rug pull risks. We run the GoPlus version V1.1.13 [43]. Pied-Piper [51] is the most recent academic tool that detects certain backdoor threats in tokens based on the EVM bytecode. Four of the backdoor threats it presents are equivalent to 2 rug pull risks (FrozeAccount for BlackList; ArbitraryTransfer, DestroyToken, and GenerateToken for ModifyBalance). We run the latest version of Pied-Piper in their open-sourced repository.

All experiments are conducted on a machine with 128 CPU cores (AMD EPYC 7742 64-Core Processor) and 512 GB memory. The OS is Ubuntu 20.04.2 LTS. We design the experiments to address the following research questions:

- **RQ1:** Can Tokeer achieve a better detection accuracy compared with state-of-the-art tools?
- **RQ2:** Is Tokeer effective in exposing hidden rug pull risks?
- **RQ3:** What is the time overhead of Tokeer?

5.1 Detection Accuracy

In this section, we evaluate the detection accuracy of Tokeer, GoPlus and Pied-Piper on the sampling dataset. We collect the results and count the rates of FP (false positive), FN (false negative) and TP (true positive) of them. We present the detailed statistics of the metrics in Table. 1. and use Fig. 8 to visualize the comparison of the accuracy between Tokeer, GoPlus, and Pied-Piper. We apply the metrics *recall* and *precision*[2]: Precision refers to the proportion of TP out of all instances predicted as positive. Recall represents the proportion of TP out of all actual positive instances.

Table 1: The false positive (FP), false negative (FN), and true positive (TP) of Tokeer, GoPlus and Pied-Piper.

Rug Pull Risks	Total	Tokeer			GoPlus			Pied-Piper		
		TP	FP	FN	TP	FP	FN	TP	FP	FN
Blacklist	134	133	0	1	52	3	79	17	1	116
ModifyBalance	118	115	1	3	66	1	51	0	18	100
TimeLimit	54	52	2	0	17	0	37	-	-	-
AlienDepend	117	117	0	0	25	0	92	-	-	-
Total / %	**423**	**98.6**	**0.7**	**0.9**	**37.8**	**0.9**	**61.2**	**6.7**	**7.5**	**85.7**

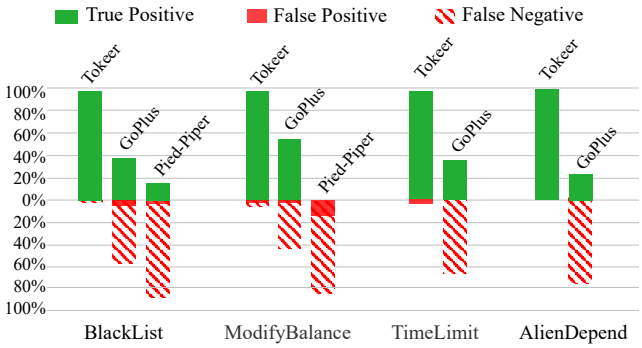

Figure 8: Comparing Tokeer to GoPlus, and Pied-Piper on sampling dataset. Tokeer achieves better detection accuracy.

For 4 rug pull risks, Tokeer achieves recalls of 99.2%, 96.6%, 96.3%, and 100.0%. These data for GoPlus are 38.8%, 55.9%, 31.5%, 21.4%, and for Pied-Piper are 12.7%, 0%, -, - (where - represents no support). Tokeer achieves a significantly higher recall than GoPlus and Pied-Piper and correctly detects the majority of risks. Moreover, Tokeer achieves precisions of 100.0%, 99.1%, 96.3%, and 100.0%. These data for Goplus are 94.5%, 98.5%, 100.0%, and 100.0%. For Pied-piper, the results are 94.4%, 0%, -, -. Tokeer has the same reliable accuracy as

[2] $recall = \frac{TP}{TP+FN}$, $precision = \frac{TP}{TP+FP}$

the commercial tool GoPlus and is significantly better than Pied-Piper. We present the detailed results and analysis in the following.

For BlackList, Tokeer successfully detects 133/134 cases. Only 1 FN EthMoon [16] is caused by using external contracts to record the BlackList so it is marked as AlienDepend. There are 79 and 166 tokens that GoPlus and Pied-Piper fail to report. For example, SWR [29] in Fig. 9 hides the BlackList in another function, which is invoked before transfer. Fite [17] adds an additional transfer switch to block the blacklisted addresses. For 3 FPs of GoPlus(e.g. GMX [15]), the mappings they defined to set the users' permissions do not really affect the transfer process. Pied-Piper falsely alerts 1 sample (HSE_Token [20]) because it misreads some operations that do not affect the transfer. They lack the effective analysis of the transfer flow and thus cannot deal with complex implementations.

```
function getCheckRoot(address sender,...) ... returns(bool statue){
   ...
   if(_isRoot[sender] || _abnormal){ return true; }
   return false;
}
function _transfer(address from, address to,...) internal { ...
   if(getCheckRoot(from,to)){return;}
   ...
}
```

Figure 9: Token SWR with BlackList hidden in complex structures that only Tokeer can detect.

For ModifyBalance, Tokeer successfully detects 115/118. Tokeer misses 3 [19, 22, 23] because they implement the transfer in external contracts with no explicitly declared balance variables. So they are marked with AlienDepend. Tokeer's 1 FP [13] is because the action of processing fee is highly similar to modifying balances in its implementation. There are 51 and 100 FNs of GoPlus and Pied-Piper such as DZD [14] in Fig. 10), which contains a transfer that only owners can call to transfer the asset to themselves. It applies source level code obfuscation that bypass the check of GoPlus. Besides, the multiple checks and external calls make the execution flow too complex to be detected by GoPlus and Pied-Piper. GoPlus has 1 FP (SQUISHY [25]) that contains a burn function that is never called and cannot cause a rug pull. GoPlus lacks the analysis of the execution flow, therefore cannot tell if this function actually makes an impact. Pied-Piper has 18 FPs (e.g. JesusCrypt [21]) because it mistakes the updates of other variables as balances.

```
constructor (...) { ...
    dzdsop[msg.sender]=true;
}
function hdzuyhxlk(uint256 amount, address ut) public {
    require(dzdsop[msg.sender], "dsacv");
    IERC20(ut).transfer(msg.sender, amount);
}
```

Figure 10: Token DZD with ModifyBalance risk. Only Tokeer can model the complex execution flow to expose the risk.

For TimeLimit, Tokeer successfully detects all cases. For 37 cases that only Tokeer can detect, 35/37 are TPs. They hide the time checks in complicated implementations that GoPlus fails to detect such as hiding the check in function invocations (e.g. SX [26]), or storing the result and checking it in the subsequent process to indirectly interrupt transfer (s.g. OXO [24] in Fig. 11). 2 FPs (107DAO [11], AntNest [12]) of Tokeer do not use time to directly restrict transfers, but for calculating random addresses. Without semantic information, Tokeer fails to decide the actual intentions of certain operations in such rare cases.

```
function _transfer(address from, address to,...) internal { ...
   if (block.timestamp - openTime < limit) {...
       data.setAddress2UintData(user, 1);
   } ...
   if (data.address2uintMapping(from) == 1) { return; }
   ...
}
```

Figure 11: OXO with TimeLimit that only Tokeer can detect. It uses the intermediate result to indirectly affect the transfer.

For AlienDepend, Tokeer successfully detects all cases, with no FP and FN. GoPlus misses 92/117 samples. The alien functions may rather implement some sub-processes during the transfer (e.g. XEN [28] in Fig. 12), or perform certain restrictions (e.g. GoldDogMoon [18]). GoPlus cannot detect them because it cannot effectively analyze the transfer execution flow and identify the effect of external operations. The addresses of the external contracts are input parameters of the constructor, or can be set only by the owner. Therefore, the external functions are invisible to users, and their security cannot be guaranteed.

```
interface dividendStaking {
   function arriveFeeRewards(uint256 amountIn) external;
}
contract XEN {
   dividendStaking public dividend;
   function setDividend(address _dividend) external onlyOwner {
      dividend = dividendStaking(_dividend);
   }
   function swapTokenToUSD(...) private {
      try dividend.arriveFeeRewards(...) {} catch {}
   }
   function _transfer(address sender, ...) internal { ...
       swapTokenToUSD(...);
       ...
   }
}
```

Figure 12: Token XEN with AlienDepend risk that only Tokeer can detect. The external call is wrapped in an outside function that cannot be detected by existing works.

In conclusion, existing works lack effective analysis of transfer execution flow and cannot identify the malicious operations hidden in complex implementations. Their oracles are not effective to cover various real-world scenarios. Tokeer establishes the transfer model to accurately analyze the transfer process, generate effective oracles, and identify the risks that can really affect the transfer. Therefore, Tokeer achieves better accuracy.

Answer to RQ1: Tokeer achieves a satisfying identification accuracy of a 98.0% reacll and a 98.9% precision, significantly better than state-of-the-art tools.

5.2 Detection Effectiveness

To evaluate the detection effectiveness of Tokeer, we first apply Tokeer, GoPlus and Pied-Piper to the survey dataset to compare them in exposing rug pull tokens that have already caused financial losses. Furthermore, we extend our analysis to encompass the full dataset to demonstrate Tokeer's effectiveness in a real-world, large-scale production environment.

Out of the total 201 rug pull tokens, Tokeer successfully detects all of them, while GoPlus and Pied-Piper detect 104 (51.7%) and 40 (19.9%) tokens, respectively. Table 2 presents the identified number of each risk on the survey dataset.

Table 2: The number of detected risks of Tokeer, GoPlus and Pied-Piper on tokens of incurred loss.

	Total	Tokeer	GoPlus	Pied-Piper
BlackList	118	118	46	17
ModifyBalance	85	85	46	4
TimeLimit	16	16	7	0
AlienDepend	50	50	3	0

Specifically, GoPlus only detects 46 (39.0%) BlackList, 46 (54.1%) ModifyBalance, 7 (43.8%) TimeLimit, and 3 (6%) AlienDepend. As for Pied-Piper, it only detects 17 (14.4%) BlackList and 4 (4.7%) ModifyBalance. The high proportion of false negatives is mainly caused by the diversity of contract code structures that cannot be effectively detected by GoPlus and Pied-Piper. For example, Ordinals Finance [36] rugged in April, 2023 through developers pulling 256 million OFI tokens [40] out to themselves, resulting in a $1 million loss. As shown in Fig. 13, `safuToken` and `ownerRewithdraw` only can be called by the owner, and can arbitrarily transfer away certain tokens to the owner's address. The malicious transfer is disguised by invoking external contracts, whose address can be manipulated by the owner. GoPlus and Pied-Piper cannot identify the complex transfer execution flow. Consequently, they cannot decide which operations threat the transfer process. In contrast, Tokeer successfully identifies the external functions that impact the transfer execution, and labels it with AlienDepend and ModifyBalance risks. Besides, some of the false negatives of GoPlus are caused by its heavy reliance on the Solidity source code. In the case of BNB42 [62] mentioned in Section 1, the source code is not public, and therefore GoPlus is unable to analyze and detect such cases.

```
constructor(address _OFI) public {
    OFI = IERC20(_OFI);
}
function ownerRewithdraw(uint256 _amount) public onlyOwner {
    ...
    OFI.transfer(msg.sender, _amount);
}
function safuToken(address token) external onlyOwner {
    if (token == address(0)) {
        payable(msg.sender).transfer(address(this).balance);
    } else {
        IERC20(token).transfer(msg.sender, IERC20(token).balanceOf(address(this)));
    }
}
```

Figure 13: Function `safuToken` and `ownerRewithdraw` in OFI.

To further assess Tokeer's detection effectiveness in a wild, large-scale real-world production environment, we run Tokeer, GoPlus, and Pied-Piper on the full dataset. Fig. 14 shows the results. Each circle represents the result set of a tool. The overlap region denotes the intersection between multiple sets, i.e., the cases that can be both detected by corresponding tools. Among 3,562 token contracts,

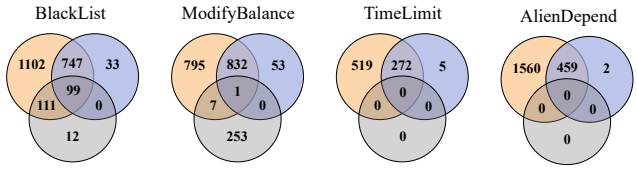

Figure 14: Number of detected cases by Tokeer, GoPlus, and Pied-Piper on 4 rug pull risks.

Tokeer detects 2,059(57.8% for BlackList), 1,635(45.9% for ModifyBalance), 791(22.2% for TimeLimit), and 2,019(56.7% for AlienDepend) risks, respectively. These data for GoPlus are 879(24.7%), 886(24.9%), 77(6.2%), 461(12.9%), and for Pied-Piper are 222(6.2%), 261(7.3%), 0(0%), and 0(0%). Tokeer detects more risks than GoPlus and Pied-Piper on all 4 rug pull risks. There are 30.9%, 22.3%, 43.8%, 14.6% tokens with rug pull risks can only be detected by Tokeer. GoPlus applies pre-defined patterns to match for known rug pull patterns in the contract. Therefore, it has a few FP, but a large amount of FN. Due to the diversity of code structures in real-world production, GoPlus's oracles are unable to detect more than half of the risks. Pied-Piper applies datalog analysis based on the contract's EVM bytecode to detect BlackList and ModifyBalance. However, it lacks the transfer modeling and effective oracles, which lead to its low detection accuracy.

The evaluation proves the effectiveness of Tokeer in real-world production. Tokeer can detect more potential rug pull risks with a very low rate of FN and FP. Tokeer applies the component-configurable transfer model to analyze the execution flow, thus ensuring that the identified risks can actually affect the real-world transfer process. Besides, the oracles generated by Tokeer are applicable to various code structures.

> **Answer to RQ2:** Tokeer is effective in exposing rug pull risks. It detects all rug pull tokens in the survey dataset, and detects 27.2% more risks in the full dataset.

5.3 Time Overhead

We collect the time overhead of Tokeer, Pied-Piper, and GoPlus on the full dataset. On average, Tokeer takes 12.64 seconds to complete the analysis for one token. Pied-Piper takes 129.4 seconds. As for GoPlus's API, given a new token, it takes more than 10 minutes to provide a result response.

Table 3: The time overhead of Tokeer.

Bytecode Size /KB	Decompilation Time /s	Detection Time /s			
		BlackList	Modify Balance	Time Limit	Alien Depend
<10	1.15	2.59	0.13	4.47	0.13
10 ~15	8.11	19.42	0.63	30.64	0.62
>15	29.74	55.55	1.11	75.73	1.09

Tokeer needs two phases to analyze a token, including the decompilation phase and the detection phase. During the decompilation,

Tokeer decompiles the runtime bytecode of the token contract and constructs the high-level 3-address IR. For detection, Tokeer uses the transfer model and oracles to conduct datalog analysis on the IR. We collect the time consumption of Tokeer on the full dataset and present the results in Table 3.

Overall, the decompilation time and detection time increase as the bytecode size increases. On average, for tokens with a bytecode size of <10KB, 10KB∼15KB, and ≥15KB, it takes Tokeer 1.15s, 8.11s, and 29.74s respectively to decompile them. The growth of decompilation time is significant because the increase of code blocks will cause an exponential increase in control flow paths, resulting in a substantial growth in the complexity of the control flow graph. The time consumption of detection is related to the oracle complexity. For BlackList and TimeLimit, the oracle consists of multiple execution flow requirements, resulting in a higher analysis complexity. For AlienDepend and ModifyBalance, time is mainly spent on transfer modeling. Their oracles are simpler because the external call statements and balance update statements are easy to locate.

Answer to RQ3: Tokeer's time overhead is relatively low compared with existing works. It consumes 1.15s∼29.74s for decompilation and 0.13s∼75.73s for detection.

6 LESSONS LEARNED

Rug pull scams are prevalent in the token market. Our evaluation 5.2 and previous studies reveals that the existence of rug pull risks is significantly more pervasive than it appears. When Xia et al. [65] identify scam tokens through history transactions, they suggest that over 50% of the tokens on Uniswap have conducted rug pull behaviors. The analysis of Cernera et al. [30] finds that 81.2% of 1-day tokens (comprising 60% of all tokens deployed) on PancakeSwap contain rug pull behavior patterns. Our results also suggest a large proportion of the newly deployed tokens with rug pull risks. However, the broad spectrum of rug pull risks does not necessarily equate to an equally diverse range of actual losses. Whether the risks result in actual harm ultimately hinges on the intentions of the developer, which others cannot ascertain. In light of this, our goal is to identify all potential risks that may result in rug pull losses and offer people an early warning system and reference points to consult before investing.

The various solidity version may affect the detection. In real-world production, the solidity version varies among different token projects. The different compiler may affect the content of EVM bytecode, which in turn affect the IR decompilation process. However, our evaluation dataset encompasses a wide range of solidity compiler versions, and we find that Tokeer consistently demonstrates high detection accuracy. Therefore, presently, variations in solidity versions do not significantly impact Tokeer's detection. If future updates do affect the detection, Tokeer may also require adaptions such as adding the new syntax in notation definitions.

7 RELATED WORK

7.1 Fraud Risks in Tokens

Detecting fraud risks in token contracts has received substantial attention due to the huge losses they inflicted on investors' assets. For traditional scams in token contracts, [34, 63, 66, 68] focus on phishing scams, [5, 35, 49] investigate Ponzi Schemes, and [4, 8, 59] explore automated scam detection. CryptoScamTracker [67] studies cryptocurrency giveaway scams and uses certificate transparency logs to identify likely giveaway scams. Pluto [52] applies inter-contract control flow graph to detect inter-contract attack scenarios.

Today, more research is focusing on new types of scams targeting the token market. Some of the studies [3, 41, 53, 64, 65] detect schemes like rug pulls based on transaction sequences. For example, Xia et al. [65] identifies scam tokens on Uniswap through machine learning and guilt-by-association heuristic on previous transactions related to Uniswap V2 exchange. Cernerma et al. [30] match the certain rug pull pattern based on the history transactions. Other tools like [1, 48, 51, 57] detect potential rug pull risks based on the source code or EVM bytecode of smart contracts. Commercial tools like GoPlus [57], TokenSniffer [48], and RugPullDetector [1] automatically audit token contracts for basic token information and potential risks. Pied-Piper [51] applies domain-specific datalog analysis to abstract the data structures and identification rules related to backdoor risks.

7.2 Traditional Vulnerability in Smart Contract

Most of the previous works related to smart contracts security focus on the detection of traditional vulnerabilities included in SWC Registry [55] such as overflow/underflow, reentrancy, and transaction order dependency. OYENTE [50] applies symbolic execution on EVM bytecode to checks for various patterns, and Osiris extends it to more arithmetic bugs. Mythril [38] integrates symbolic execution, SMT solving and taint analysis to detect a variety of traditional security vulnerabilities. Besides, formal verification is also widely used for more reliable results. ZEUS [47] applies automatic formal verification of smart contracts using abstract interpretation and symbolic model checking to detect bugs like reentrancy and overflow. Securify [61], a security analyzer for Ethereum smart contracts, uses datalog to express analysis computations in its fact inference engine. MadMax [45] also uses datalog analysis, but detect gas-focused vulnerabilities within smart contracts.

7.3 Main Difference

Tokeer is a token verification tool that specifically focuses on the new prevalent scamming approach: rug pull. Its main objective is to identify the injected rug pull risks in the token contracts and provide early alerts to investors of potential loss. Compared to existing tools, Tokeer stands out with its transfer model and oracle generation strategy that overcome the interference of diverse code structures. This allows Tokeer to achieve a stronger ability to uncover hidden risks and provide more effective risk management for investors.

8 CONCLUSION

In this paper, we conduct an in-depth study on 201 rug pull events and summarize 4 types of rug pull risks. We propose a component-configurable transfer model and oracle generation strategy. We implement them and propose Tokeer to detect the rug pull risks in crypto tokens. For evaluation, we compare Tokeer with Pied-Piper and a commercial tool GoPlus on large scale real-world tokens. The results show that Tokeer outperforms GoPlus and Pied-Piper

in terms of detection accuracy, with a 98.0% recall and a 98.9% precision on average. Tokeer exposes 27.2% more real rug pull risks than state-of-the-art tools. Our future work will focus on exploring more rug pull patterns and conducting more evaluations on new-employed tokens.

9 ACKNOWLEDGEMENT

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000) and NSFC Program (No. 62022046, 92167101, 62021002).

REFERENCES

[1] Rug pull detector. http://rugpulldetector.com/. Accessed at June 19, 2023.
[2] Beosin Alert. Jst. https://twitter.com/BeosinAlert/status/1579058826774343680, 2022. Accessed at June 19, 2023.
[3] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. Clockwork finance: Automated analysis of economic security in smart contracts. *arXiv preprint arXiv:2109.04347*, 2021.
[4] Emad Badawi, Guy-Vincent Jourdan, Gregor Bochmann, and Iosif-Viorel Onut. An automatic detection and analysis of the bitcoin generator scam. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 407–416. IEEE, 2020.
[5] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *Future Generation Computer Systems*, 102:259–277, 2020.
[6] Beosin. Securing blockchain ecosystem. https://beosin.com, 2023. Accessed at June 19, 2023.
[7] BeosinAlert. Nova exchange on bsc rugged. https://twitter.com/BeosinAlert/status/1601168659585454081, 2022. Accessed at June 19, 2023.
[8] Lingyu Bian, Linlin Zhang, Kai Zhao, Hao Wang, and Shengjia Gong. Image-based scam detection method using an attention capsule network. *IEEE Access*, 9:33654–33665, 2021.
[9] Blocksec. Building blockchain security infrastructure. https://blocksec.com/, 2023. Accessed at June 20, 2023.
[10] BlockSecTeam. A rug pull of 2m busd. https://twitter.com/BlockSecTeam/status/1613492776712249344. Accessed at June 20, 2023.
[11] BSCscan. 107dao. https://bscscan.com/address/0x01a5eab27481f7382a62af57ca37d6972bbab21f#code, 2022. Accessed at June 18, 2023.
[12] BSCscan. Antnest. https://bscscan.com/address/0x131a1c7196a5ca7c9f3f8ad123a66bf0becc1a8f#code, 2022. Accessed at June 20, 2023.
[13] BSCscan. Ducktoken. https://bscscan.com/address/0x437d7cb2ba0e73fdffd4e475b18bae0d40a9346f#code, 2022. Accessed at June 20, 2023.
[14] BSCscan. Dzd. https://bscscan.com/address/0x644504097ea250016df3cd7f350004b44de1d56f#code, 2022. Accessed at June 20, 2023.
[15] BSCscan. Erc20token. https://bscscan.com/address/0x27800b47F63966fBcE696CDD683DF3A2FFf82261#code, 2022. Accessed at June 18, 2023.
[16] BSCscan. Ethmoon. https://bscscan.com/address/0xe493f107bdba58fcb45bce7e5ed753272edd6361#code, 2022. Accessed at June 19, 2023.
[17] BSCscan. Fite. https://bscscan.com/address/0x76347d0192cb045abc42166f931988e2566b43cf#code, 2022. Accessed at June 20, 2023.
[18] BSCscan. Golddogmoon. https://bscscan.com/address/0xc914bd429f00da7e7b09f63cbc61b479ed4e86bf#code, 2022. Accessed at June 18, 2023.
[19] BSCscan. Hao. https://bscscan.com/address/0x363d69655d63ad19877b841ebe424f3e010b347f#code, 2022. Accessed at June 20, 2023.
[20] BSCscan. Hsetoken. https://bscscan.com/address/0x141553ba94b9155be1e1e9882f60a131a990521f#code, 2022. Accessed at June 19, 2023.
[21] BSCscan. Jesuscrypt. https://bscscan.com/address/0x1b8b8a61d76a2ada501544b7b936a877ef92dd5f#code, 2022. Accessed at June 20, 2023.
[22] BSCscan. Kicks. https://bscscan.com/address/0xa8f49b2f0e96dc0b4194f0c7fb191b4eddedbc5f#code, 2022. Accessed at June 19, 2023.
[23] BSCscan. Leap. https://bscscan.com/address/0xa8f49b2f0e96dc0b4194f0c7fb191b4eddedbc5f#code, 2022. Accessed at June 18, 2023.
[24] BSCscan. Oxo. https://bscscan.com/address/0x29f6b1b7f024752fae51a83c0515fe469508084f#code, 2022. Accessed at June 18, 2023.
[25] BSCscan. Squishy. https://bscscan.com/address/0xacb0dd43e0c34774581e4c4a579086baeabf009f#code, 2022. Accessed at June 18, 2023.
[26] BSCscan. Sx. https://bscscan.com/address/0x1676341ce18f2f01c5d32b84f8e48e28494f7f6f#code, 2022. Accessed at June 20, 2023.
[27] BSCscan. Token. https://bscscan.com/address/0x6dfb288bb8040bb7d63b51f56bdce58bdfe87c10#code, 2022. Accessed at June 19, 2023.
[28] BSCscan. Xen. https://bscscan.com/address/0x6514c28c54dc24b30f7b0ac204e116e41264112f#code, 2022. Accessed at June 18, 2023.
[29] Bscscan. Swr. https://bscscan.com/address/0x45764a1e56a58f8f074a133785225a8595c91d2f#code, 2023. Accessed at June 19, 2023.
[30] Federico Cernera, Massimo La Morgia, Alessandro Mei, and Francesco Sassi. Token spammers, rug pulls, and sniperbots: An analysis of the ecosystem of tokens in ethereum and the binance smart chain (bnb). *arXiv preprint arXiv:2206.08202*, 2022.
[31] CertiK. Securing the web3 world. https://www.certik.com, 2023. Accessed at June 19, 2023.
[32] BNB Smart Chain. Bscscan. https://bscscan.com, 2023. Accessed at June 20, 2023.
[33] chainalysis. The chainalysis 2023 crypto crime report. https://go.chainalysis.com/2023-crypto-crime-report.html, 2023. Accessed at June 20, 2023.
[34] Weili Chen, Xiongfeng Guo, Zhiguang Chen, Zibin Zheng, and Yutong Lu. Phishing scam detection on ethereum: Towards financial security for blockchain ecosystem. In *IJCAI*, pages 4506–4512, 2020.
[35] Weili Chen, Zibin Zheng, Edith C-H Ngai, Peilin Zheng, and Yuren Zhou. Exploiting blockchain data to detect smart ponzi schemes on ethereum. *IEEE Access*, 7:37575–37586, 2019.
[36] cointelegraph. Ordinals finance has conducted a $1m rug pull: Certik. https://cointelegraph.com/news/ordinals-finance-has-conducted-a-1m-rug-pull-certik, 2023.
[37] Comparitech. Worldwide crypto & nft rug pulls and scams tracker. https://www.comparitech.com/crypto/cryptocurrency-scams/, 2023. Accessed at June 19, 2023.
[38] ConsenSys. Mythril. https://github.com/ConsenSys/mythril-classic, 2018.
[39] Ethereum. Ethereum/solidity. https://github.com/ethereum/solidity. Accessed at June 19, 2023.
[40] Etherscan. Oebstaking. https://etherscan.io/address/0x4d8266ec8ded77edac50a0eefe3d6934b53663cd#code, 2023.
[41] Shuhui Fan, Shaojing Fu, Yuchuan Luo, Haoran Xu, Xuyun Zhang, and Ming Xu. Smart contract scams detection with topological data analysis on account interaction. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 468–477, 2022.
[42] Sirius Finance. Sirius finance. https://twitter.com/realpolkabridge/status/1560442022779310080. Accessed at June 20, 2023.
[43] GoPlus. Change logs. https://docs.gopluslabs.io/reference/token-security-api-response-detail/change-logs, 2022. Accessed at June 18, 2023.
[44] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.
[45] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.
[46] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.
[47] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.
[48] Solidus Labs. Token sniffer. https://tokensniffer.com. Accessed at June 18, 2023.
[49] Daniel Liebau and Patrick Schueffel. Crypto-currencies and icos: Are they scams? an empirical study. *An Empirical Study (January 23, 2019)*, 2019.
[50] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.
[51] Fuchen Ma, Meng Ren, Lerong Ouyang, Yuanliang Chen, Juan Zhu, Ting Chen, Yingli Zheng, Xiao Dai, Yu Jiang, and Jiaguang Sun. Pied-piper: Revealing the backdoor threats in ethereum erc token contracts. *ACM Transactions on Software Engineering and Methodology*, 2022.

[52] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering*, 48(11):4380–4396, 2021.

[53] Bruno Mazorra, Victor Adan, and Vanesa Daza. Do not rug on me: Leveraging machine learning techniques for automated scam detection. *Mathematics*, 10(6):949, 2022.

[54] PeckShield. A blockchain security and data analytics company. https://twitter.com/peckshield, 2023. Accessed at June 20, 2023.

[55] SWC Registry. Smart contract weakness classification and test cases. https://swcregistry.io, 2020. Accessed at June 20, 2023.

[56] Rugdoc.io. Tomb fork rug alert. https://twitter.com/RugDocIO/status/1531672289346977793, 2023. Accessed at June 20, 2023.

[57] GoPlus Security. Goplus security. https://goplus labs.io. Accessed at June 18, 2023.

[58] Soliduslabs. The 2022 rug pull report. https://www.soliduslabs.com/reports/rug-pull-report, 2023. Accessed at June 20, 2023.

[59] Patel Nikunjkumar Sureshbhai, Pronaya Bhattacharya, and Sudeep Tanwar. Karuna: A blockchain-based sentiment analysis framework for fraud cryptocurrency schemes. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2020.

[60] TMK. Tmk. https://bscscan.com/token/0x47e79481c404f295b292096bb50373eef97d06ff#code. Accessed at June 18, 2023.

[61] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[62] web3isgreat. Bnb42 rug pulls for over $2.7 million. https://twitter.com/web3isgreat/status/1498420099815985153, 2022. Accessed at June 18, 2023.

[63] Jiajing Wu, Qi Yuan, Dan Lin, Wei You, Weili Chen, Chuan Chen, and Zibin Zheng. Who are the phishers? phishing scam detection on ethereum via network embedding. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2020.

[64] Pengcheng Xia, Bingyu Gao, Weihang Su, Zhou Yu, Xiapu Luo, Chao Zhang, Xusheng Xiao, Guoai Xu, et al. Demystifying scam tokens on uniswap decentralized exchange. *arXiv preprint arXiv:2109.00229*, 2021.

[65] Pengcheng Xia, Haoyu Wang, Bingyu Gao, Weihang Su, Zhou Yu, Xiapu Luo, Chao Zhang, Xusheng Xiao, and Guoai Xu. Trade or trick? detecting and characterizing scam tokens on uniswap decentralized exchange. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(3), dec 2021.

[66] Pengcheng Xia, Haoyu Wang, Xiapu Luo, Lei Wu, Yajin Zhou, Guangdong Bai, Guoai Xu, Gang Huang, and Xuanzhe Liu. Don't fish in troubled waters! characterizing coronavirus-themed cryptocurrency scams. In *2020 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–14. IEEE, 2020.

[67] Nick Nikiforakis Xigao Li, Anurag Yepuri. Double and nothing: Understanding and detecting cryptocurrency giveaway scams. In *NDSS*, 2023.

[68] Qi Yuan, Baoying Huang, Jie Zhang, Jiajing Wu, Haonan Zhang, and Xi Zhang. Detecting phishing scams on ethereum based on transaction records. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.