Automated Software Engineering (2024) 31:63
https://doi.org/10.1007/s10515-024-00459-4

Angels or demons: investigating and detecting decentralized financial traps on ethereum smart contracts

Jiachi Chen[1] · Jiang Hu[1] · Xin Xia[2] · David Lo[3] · John Grundy[4] · Zhipeng Gao[5] · Ting Chen[6]

Received: 29 November 2023 / Accepted: 7 July 2024

Abstract
Decentralized Finance (DeFi) uses blockchain technologies to transform traditional financial activities into decentralized platforms that run without intermediaries and centralized institutions. Smart contracts are programs that run on the blockchain, and by utilizing smart contracts, developers can more easily develop DeFi applications. Some key features of smart contracts—self-executed and immutability—ensure the trustworthiness, transparency and efficiency of DeFi applications and have led to a fast-growing DeFi market. However, misbehaving developers can add traps or backdoor code snippets to a smart contract, which are hard for contract users to discover. We call these code snippets in a DeFi smart contract as "*DeFi Contract Traps*" (DCTs). In this paper, we identify five DeFi contract traps and introduce their behaviors, describe how attackers use them to make unfair profits and analyze their prevalence in the Ethereum platform. We propose a symbolic execution tool, DeFiDefender, to detect such traps and use a manually labeled small-scale dataset that consists of 700 smart contracts to evaluate it. Our results show that our tool is not only highly effective but also highly efficient.DeFiDefender only needs 0.48 s to analyze one DeFi smart contract and obtains a high average accuracy (98.17%), precision (99.74%)and recall (89.24%). Among the five DeFi contract traps introduced in this paper, four of them can be detected through contract bytecode without the need for source code. We also apply DeFiDefender to a large-scale dataset that consists of 20,679 real DeFi-related Ethereum smart contracts. We found that 52.13% of these DeFi smart contracts contain at least one contract trap. Although a smart contract that contains contract traps is not necessarily malicious, our finding suggests that DeFi-related contracts have many centralized issues in a zero-trust environment and in the absence of a trusted party.

Keywords Smart contract analysis · Ethereum · Decentralized financial · Financial traps

Extended author information available on the last page of the article

1 Introduction

In traditional financial systems, centralized institutions ensure the security of financial activities. However, it is not easy to guarantee the reliability and trustworthiness of the centralized node. For example, a peer-to-peer lending application claims high interest for the investors, but investors may worry about whether the P2P application will pay the interest as they promised, even return the principal. Thus, traditional financial institutions will often encounter frictions in the fund raising process, especially for some infamous enterprises/institutions, as they have no strong network ties to make investors trust them Chen and Bellavitis (2020).

Fortunately, the advent of Ethereum smart contracts[1] allows enterprises and institutions to address the trust concern and also brings other new advantages. First, smart contracts are self-executed—all of their execution depends on their code. If there is no code related to a money transfer, even the contract owner cannot withdraw the money. Second, smart contracts are immutable. Once the smart contracts are deployed to the blockchain, it is very difficult to change their code or behavior. These features help to ensure the *trustworthiness* of smart contracts. Third, all smart contract transactions on Ethereum are visible to anyone (Chen et al. 2018), which ensures their *transparency*. Applications based on smart contracts are able to replace some trust requirements, which makes them more *efficient* than traditional financial systems. Specifically, money transfer between different countries or institutions may need several hours, even days, in traditional financial systems because of security policies, while token transfer in smart contract-based applications usually only needs several minutes (Schär 2020).

The advent of smart contracts results in the derivation of a new term, so-called "*Decentralized Finance*,"[2] also known as "DeFi". In DeFi, traditional financial activities are transformed into decentralized platforms by utilizing decentralized networks or technologies, e.g., smart contracts. The features of smart contracts, i.e., *trustworthiness, transparency* and *efficiency*, lead to the fast-growing of DeFi ecosystem. In this paper, we refer to all financial activities rooted in smart contracts as "DeFi Contracts".

As the most popular blockchain platform to run smart contracts (Chen et al. 2019), the number of DeFi applications deployed on Ethereum has been growing rapidly in recent years. More than 100 billion dollars of value is locked in the Ethereum in 2021[3] with a wide variety of DeFi applications, e.g., peer-to-peer lending,[4] token exchanges.[5]

[1] Ethereum.org. https://www.ethereum.org/.

[2] Peer to Peer Lending Blockchain Platform. https://consensys.net/blockchain-use-cases/decentralized-finance/.

[3] Total Value Locked in Ethereum. https://defillama.com/chain/Ethereum

[4] Blockchain for Decentralized Finance (DeFi). https://consensys.net/blockchain-use-cases/decentralized-finance/

[5] ERC20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

However, it is inevitable that a new finance-oriented technology might be exploited by misbehaving developers. A DeFi application usually has a front end to interact with users, e.g., a website or mobile app. In the front end, users are informed that the application is based on smart contracts to ensure its trustworthiness. This front end usually provides links to smart contracts, which enable users to check the information of contracts, e.g., transactions and balances. The *immutable, self-executed* and *public information* of smart contracts might give users a false sense of security that the DeFi applications cannot take advantage of their investments. It is true that smart contracts are immutable and self-executed. However, misbehaving developers can add some traps or backdoors in their smart contract code to unfairly profit from users. Many smart contracts do not come with publicly available source code (Chen et al. 2019, 2020), and Ethereum itself only stores the bytecode of smart contracts. Without source code, even professional smart contract developers find it difficult to spot DeFi traps from contract bytecode. Some smart contract source codes might be visible to the public. However, many users may not have enough professional knowledge to see through the DeFi traps added to the code. Thus, misbehaving developers can make an unfair profit from users by adding traps or backdoors in the DeFi smart contract code.

In this paper, we introduce the concept of *DeFi Contract Traps* (DCTs); we define a DCT as “A code snippet in a DeFi smart contract that *can* allow unscrupulous contract owners to unfairly profit from unsuspecting contract users.” There is a key difference between DCTs and vulnerabilities in traditional software code: vulnerabilities are program “errors” that can lead to security issues. Vulnerabilities can be activated by anyone and might lead to financial loss for both contract owners and users. However, DCTs are more like “warnings” for contract users and can only be executed by contract owners. That is to say, contracts with DCTs are not necessarily malicious. DCTs give contract owners the ability to make unfair profits from contract users. DCTs thus pose threats to contract users’ assets, while potentially unfairly benefitting contract owners, but may not be used by the contract owners. In some situations, vulnerabilities could transform to DCTs by adding additional conditions (code snippets). These kinds of DCTs could be regarded as the “transformation” of vulnerabilities (cf. Sect. 6.4).

To help readers better understand the concept of DCTs, we introduce five DCTs as examples and introduce how attackers can use these traps to make unfair profits. Two of them—*Tricky Send* and *Selfdestruct Permission*—are the “transformation” of vulnerabilities. *Super Storage Permission* and *Super Transfer Permission* are suspicious code snippets that enable DeFi smart contract owners to transfer money or change the states of the contracts. *Forged Transfer* cheats users by adding promised functionalities to unexecuted parts of a contract.

To combat these DeFi contract traps, we propose a new tool, DeFiDefender, to detect defined DCTs. There are two parts to DeFiDefender—a symbolic executor and source analyzer. For *Forged Transfer*, we have to detect it from smart contract source code by parsing its Abstract Syntax Tree (AST). For the other four DeFi traps, DeFiDefender is able to detect them through contract bytecode without the need for their contracts’ source code. DeFiDefender uses a symbolic execution model and several predefined patterns to detect contract traps.

We use a manually labeled small-scale dataset, which consists of 700 smart contracts to evaluate DeFiDefender. Our experimental results show that DeFiDefender is not only highly accurate (achieves an average accuracy and F1-score of 98.17% and 93.85%, respectively), but is also highly efficient (only needs 0.48 s to analyze one smart contract). We also use a large-scale dataset, which consists of 20,679 real DeFi-related Ethereum smart contracts extracted from 117,926 verified (open-sourced) smart contracts on Etherscan by February 2022. Our detection results show that about 52.13% of DeFi smart contracts contain at least one DCT. Although a smart contract that contains contract traps is not necessarily malicious, it still shows that the contract owner has the ability to make an unfair profit. The main contributions of this paper include:

- We introduce the DCT warning concept and investigate the safety of DeFi contracts on Ethereum. By understanding such DCTs, contract users might realize some DeFi contracts are not as safe as they are claimed to be.
- We propose a tool named DeFiDefender to detect these DCTs. DeFiDefender obtains a high average accuracy (98.17%), precision (99.74%), and recall (89.24%), and is highly efficient (0.48 s) in analyzing one smart contract, which can be used by contract users or institutions to find malicious DeFi apps.
- We have run DeFiDefender on a large number of real Ethereum DeFi smart contracts and confirm that our defined five DCTs are prevalent in Ethereum.
- We have released the DeFiDefender tool and dataset at: https://github.com/DefiDefender/DefiDefender.

The organization of the rest of this paper is as follows. In Sect. 2, we present the background knowledge of smart contracts and DeFi. Then, we introduce five DeFi contract traps in Sect. 3 and describe the methodology of DeFiDefender in Sect. 4. After that, we present its evaluation in Sect. 5. We present a case study and discuss the importance of DCTs in Sect. 6. Then, we introduce related works in Sect. 6. In Sect. 7, we conclude the paper and present future directions.

2 Background

In this section, we briefly introduce key background information about DeFi, smart contracts, and EVM bytecode.

2.1 Decentralized finance

Decentralized finance (DeFi) refers to a financial application that is built at the top of the blockchain system, which has increased rapidly in recent years (Popescu et al. 2020). The blockchain system provides easy access with near-instant, permissionless, and transparent financial services to developers and users. DeFi applications do not need any intermediaries. The total value locked in the DeFi system was only 600 million USD in January 2020, while the number exploded to 160 billion in

December 2021. Werner et al. (2021). The explosive growth of the DeFi ecosystem is inseparable from some killer applications, e.g., Automated Market Maker (AMM) (Lin et al. 2019), Flash Loan (Gudgeon et al. 2020), and Non-Fungible Tokens (NFTs) (Wang et al. 2021). AMM uses predefined rules to enable on-chain decentralized exchanges to replace traditional order book systems. Flash loan is a type of uncollateralized lending based on smart contracts. NFTs can be associated with virtual or real-world items, e.g., photos and videos. Thus, the ownership of a real-world item can be separated into several tokens, which can be transferred or sold. There are many other applications of DeFi e.g., P2P-lending, saving applications, etc.

2.2 Ethereum and smart contracts

The concept of smart contracts was first introduced by Nick Szabo in 1997 (Szabo 1997). He described smart contracts as trustless and self-executing programs that can facilitate digital verification. In 2015, the birth of Ethereum made the concept of smart contracts become a reality.

There are two types of accounts in Ethereum, i.e., externally owned accounts (EOA) and contract accounts. For an EOA, users use the private key to control the contract. For a contract account, all the execution depends on its code; no one can affect the running of the contract, even the owner. Ether (ETH) is the token in the Ethereum platform. In August 2021, the value of a single Ether is equivalent to 3300 USD.

Smart contracts are usually developed in a high-level programming language, e.g., Vyper,[6] Solidity,[7] LLL.[8] Solidity is the most popular language in programming Ethereum smart contracts (Chen et al. 2018, 2020). Thus, we only focus on Solidity smart contracts in the Ethereum ecosystem in this paper. There are four methods to transfer Ethers in Ethereum smart contracts. Three of them are *address* related functions, i.e., *address.transfer(), address.send()*, and *address.call.value()()*. For ease of reference, we called them AT, AS, and ACV, respectively. AT and AS can only transfer Ethers to an EOA, while ACV can transfer Ethers to both two types of accounts. AS and ACV return a boolean value to identify whether the transfer is successful, while AT will throw an exception if the transfer fails. Another method to transfer Ethers is by using *selfdestruct(address)* function. This function is the only way to remove a smart contract from Ethereum. Once this method is executed, all the Ethers on a contract will be transferred to a specific address, and the contract will be destroyed.

[6] Vyper. https://vyper.readthedocs.io/.

[7] Solidity Document. http://solidity.readthedocs.io.

[8] LLL. https://lll-docs.readthedocs.io/.

2.3 Ethereum virtual machine (EVM)

Ethereum blockchain is a decentralized network that consists of distributed nodes. Each node stores the whole blockchain, which is called a distributed ledger. The node allows users to create or invoke smart contracts by sending transactions. These transactions will be processed on a node during the verification of blocks by using the Ethereum Virtual Machine (EVM).

When deploying a smart contract to Ethereum, the contract will be compiled into the EVM bytecode. There are two kinds of EVM bytecode, i.e., runtime bytecode and creation bytecode. The creation bytecode contains the constructor logic and constructor parameters of a smart contract, while the runtime bytecode stores the execution logic of a smart contract. Once a contract is deployed, Ethereum only stores the runtime bytecode.

When a user submits a transaction to invoke a contract, EVM splits bytecode into bytes and executes them one by one. Each byte represents a single EVM instruction. There are 140 unique instructions of EVM by February 2022 (Ethereum Foundation 2014). EVM is a stack-based machine, which is similar to JVM, but they have many differences. For example, JVM bytecode has a clearly defined jump target, while the jump targets of EVM are read from the EVM stack. This feature increases the difficulty of building a Control Flow Graph (CFG) from a smart contract.

2.4 Verified smart contracts in Etherscan

EtherScan[9] is the most popular block explorer to search Ethereum blockchain data. It facilitates users to check the information of the Ethereum blockchain, e.g., contract bytecode, contract balance, and transactions. Ethereum only stores the bytecode of smart contracts, and Etherscan provides a platform to allow developers to upload the source code of contracts. To upload the source code on Etherscan, developers should first inform the compiler version and give the contract name of a specific contract address. Then, Etherscan will compile the uploaded source code to bytecode locally and compare it with the bytecode stored on the blockchain. If they are the same, the source code will be public on Etherscan and can be checked by anyone. However, when compiling source code to bytecode, the compiler will remove unused parts, which means even the bytecode of two contracts are the same, the source code may not be the same. Since the Etherscan verification system of Etherscan cannot identify the unused parts on the source code and remove them, the source code published on Etherscan might contain unexecuted parts.

3 DeFi contract traps

[9] EtherScan. https://etherscan.io/

A "DeFi contract trap" refers to a code snippet in a DeFi smart contract that can allow unscrupulous contract owners to unfairly profit from unsuspecting contract users. Based on our experience in the field of DeFi smart contracts on Ethereum, we have identified five examples of DCTs. Table 1 shows a summary of their definitions, and we give detailed descriptions in the following subsections. The five traps are divided into two groups, i.e., executed contract traps and non-executed contract traps. For each trap, we give an example to introduce how misbehaving developers use it to make unfair profit.

```
pragma solidity ^0.4.25;
contract P2PLending{
  address owner;
  uint baseInterest = 100;
  mapping(address => uint) public bankAccount;
  mapping(address => uint) public startTime;
  modifier onlyOwner(){
    require(msg.sender == owner);
    _;
  }
  constructor() public{
    owner = msg.sender;
  }
  function invest() public payable {
    bankAccount[msg.sender] += msg.value;
    startTime[msg.sender] = block.number;
  }
  //Tricky Ether Transfer
  function TrickyWithdraw() public payable {
    require(bankAccount[msg.sender] > 0);
    require(block.number - startTime[msg.sender] > 1000000);
    uint bonus = bankAccount[msg.sender] * baseInterest / 1000;
    msg.sender.send(bankAccount[msg.sender] + bonus);
    bankAccount[msg.sender] = 0;
  }
  //Super Storage Permission
  function changeInterest(uint newInterest) public onlyOwner{
    baseInterest = newInterest;
  }
  //Selfdestruct Permission
  function destructContract(address addr) public onlyOwner{
    selfdestruct(addr);
  }
  //Super Transfer Permission
  function withDrawEthersByOwner(address addr, uint amount) public payable
      onlyOwner{
    if(address(this).balance > amount)
      addr.transfer(amount);
  }}
```

Listing 1 Executed Contract Traps Example - A P2P Lending Contract

3.1 Executed contract traps

Executed contract traps refer to code snippets that are utilized by misbehaving developers to generate unfair profits, and the code snippets will be compiled into contract bytecode. There are four kinds of executed contract traps introduced in this paper – *Tricky Send, Super Storage Permission, Super Transfer Permission*, and *Selfdestruct Permission*.

Table 1 Definitions of our Five Defi Contract Traps

Contract defect	Definition
Tricky send (TS)	Resetting users' balance deliberately when Ether transfer fails
Super Storage Permission (SSP)	Adding backdoor code snippets to allow someone to change key storage variables freely
Super Transfer Permission (STP)	Adding suspicious code snippets to allow someone to transfer the Ethers freely
Selfdestruct permission (SP)	Adding a Selfdestruct function to allow someone to destroy the contract and transfer all the Ethers
Forged transfer (FT)	Transferring promised functionalities to the unexecuted parts of source code to confuse users

3.1.1 Contract example

Listing 1 shows a P2P Lending contract, which claims users can invest Ethers to the contracts, and they can get 10% interest after a certain period. L11 is the constructor function of the contract. It is only executed once when the contract is deployed to the blockchain. *msg.sender* is the address of the transaction sender. Thus, the variable *owner* in L3 stores the address of the contract owner (the person who deploys the contract). A function can only be executed when it passes the logic check of modifier (L7). Thus,the function *changeInterest()* (L27), *destructContract()* (L31), and *withDrawEthersByOwner()*(L35) can only be executed by the contract owner. Users can call *invest()* (L14) to invest Ethers. Their address, investment amounts, and time of the investment will be stored (L5 and L6). After a time of period (L21), the users are able to withdraw their Ethers with 10% of interests(L23). After that, their balance will be reset (L24).

3.1.2 Tricky send (TS)

As introduced in Sect. 2, Ethereum allows three methods to transfer Ethers without destructing the contracts, i.e., *address.transfer(), address.send()* and *address.call.value()()*. Unlike *address.transfer()* which will throw an exception and stop the transaction when the Ether transfer failed, *address.send()* and *address.call.value()()* will only return a boolean value *false*. Misbehaving developers might not check the return value deliberately and reset users' balance to make unfair profit. It is difficult for users to find the risk, if they do not have enough knowledge in developing smart contracts.

Attack example The Ether transfer failure can frequently happen if the contract does not have enough balance to send. In Listing 1, the contract promises to give users 10% interest. However, there is no initial balance on the contract, which means the first user who withdraws money will definitely fail. Specifically, a user transfers 1 Ether to the contract, and the contract promises to return 1.1

Ethers. However, the insufficient balance will *lead to the failure of transfer, and the users' balance will be cleared.*

3.1.3 Super storage permission (SSP)

Immutability is an important feature of smart contract applications. Many DeFi contracts claim their safety and trustworthiness because smart contracts cannot be modified once deployed. However, the states of storage variables can be changed to allow the contract owner to control the contract. Although all the data of a smart contract is publicly viewable, it does not imply that checking the data is easy for everyone. To check the storage values, users need to use RPC interfaces, e.g., interfaces provided by web3,[10] or find the transaction that modifies the storage variables. However, both of them might not easy to be operated by smart contract users who may not be an expert in reading smart contract code. *Users may not easily realize that they are cheated by the contracts when the contract owner changes the storage values.*

Attack example Listing 1 uses a high interest (10%) to lure users to invest in the contracts. However, in Line 27, the contract owner can easily change the interest. Many users *may not be aware that the interest has been changed until they withdraw Ethers*.

3.1.4 Super transfer permission (STP)

With this DCT, *misbehaving developers can embed suspicious code in a contract.* This enables them to transfer Ethers freely, which threatens the DeFi contract users' assets.

Attack Example The *withDrawEthersByOwner()* in L35 of Listing 1 is the suspicious code added by the owner. By using this function, the contract owner can transfer Ethers which are stored by the contract users.

3.1.5 Selfdestruct permission (SP)

Selfdestruct function is the only way to disable a smart contract in Ethereum. Once this function is executed, the contract cannot be visited anymore, and all the Ethers on the contract will be transfered to another account. Usually, this function is used to prevent unexpected situations. For example, when a bug is found, the contract owner can destroy the contract and deploy a new version after fixing the bug. However, *some misbehaving developers might add a* Selfdestruct function to destruct the contract and take all of the Ethers on its balance.

Attack Example L32 of Listing 1 contains a *Selfdestruct* function, which can only be executed by the contract owner. Thus, the contract owner *has the ability to destroy the contract and transfer all the Ethers to their own account*.

[10] Web3. https://web3js.readthedocs.io/.

3.2 Non-executed contract traps

Non-executed contract traps correspond to cases where the code snippets that are used by misbehaving developers to make unfair profit will not be executed, and the code snippets cannot be found in the contract bytecode.

3.2.1 Contract example

Listing 2 is a contract with unexecuted subcontracts. The main contract is named "*Maincontract*"; *Subcontract1* is inherited by *Maincontract*, while *Subcontract2* is never executed.

```
1 contract Maincontract is subcontract1 { . . .
2 }
3 contract Subcontract1 {
4     function invest() {...}
5 }
6 contract Subcontract2 {
7     function withDrawEthers() {...}}
```

Listing 2 Non-Executed Contract Trap Example - A Contract with Unexecuted Subcontracts

3.2.2 Forged Transfer (FT)

The Etherscan verification system will not check the unexecuted parts of a smart contract (See Sect. 2.4). *Misbehaving developers can promise some functionalities to contract users to gain their trust*, such as Ether transfer or pay interest. However, they can move the promised functions to ***unexecuted*** parts to mislead users.

Attack example Listing 2 is an example of a contract with forged transfer. There is one main contract and two subcontracts in the contract. The *invest()* (L4) allows users to invest the contracts, and *withDrawEthers* (L7) allows users to withdraw Ethers. All the subcontracts are benign, and there are no special permissions for the contract owner. However, the main contract only inherits *Subcontract1*, which means *Subcontract2* can never be executed. Thus, ***the users can never withdraw the Ethers stored on the contract***.

4 The DeFiDefender approach

We introduce a new approach packaged in our DeFiDefender tool to detect the five DCTs shown above. We first explain the workflow of our DeFiDefender tool in Sect. 4.1. Then, we describe the technical details in the following parts.

4.1 Design overview

Figure 1 shows an overview of the architecture of DeFiDefender. The tool consists of two parts: the top half of the figure is used to detect four DCTs, i.e., *TS, SSP, STP*, and *SP*. DeFiDefender is able to detect these four DCTs through contract bytecode without the need for source code. If the input is source code, DeFiDefender will compile the source code to bytecode by using the Solidity compiler. After that, the bytecode is disassembled to opcode by using a method provided by Geth.[11] Next, the opcode will be split into creation bytecode and runtime bytecode according to the positions of two consecutive instructions, i.e., *RETURN STOP*. Then, we symbolically execute the creation bytecode and runtime bytecode, respectively. During symbolic execution, we record the storage slots and contract permission owners from the creation bytecode. The control flow graph (CFG), state events, and several target instructions are the by-products of the symbolic execution of runtime bytecode (see Sect. 4.3 for details). Finally, we design four patterns to detect the four DCTs based on the obtained information.

The bottom half of the figure is used to detect *Forged Transfer*. Since all the contracts with FT must have source code, DeFiDefender only supports feeding source code as input to detect it.

4.2 Symbolic execution for smart contracts

The Ethereum Virtual Machine (EVM) operates as a stack-based computing engine. During symbolic execution (SE), each instruction is processed by the SE engine, interacting with a simulated EVM stack. Using a bytecode snippet 0x6040600102 as an example. The bytecode is first split into several bytes, i.e., 0x60, 0x40, 0x60, 0x01, 0x02, and the SE engine executes them sequentially. Each byte is a hexadecimal value that represents a single instruction, and all of them can be found at Ethereum yellow paper.[12] The first byte 0x60 represents the instruction "*PUSH1*", which pushes the next one byte item into the EVM stack. Thus, 0x40 is pushed into the EVM stack. Similarly, 0x01 is also pushed into the stack. After that, 0x02, which represents the instruction "*MUL*" is executed. "*MUL*" reads two values from the stack and pushes their multiplication (0x40) back to the stack. Finally, only a value 0x40 remains on the stack. Note that in certain situations, it is not possible to directly obtain the detailed values of a variable. In such cases, we use a symbol to represent its potential value. For example, consider a function "function getBalance(Address addr)". If the specific value of *addr* is unknown, we introduce a symbol onto the stack to represent the value of the variable.

To construct a control flow graph (CFG), it is necessary to establish edges between basic blocks. A basic block is defined as a sequence of instructions that has no branching, except at the entry and exit points (Chen et al. 2020). Basic blocks are delineated by key instructions that signify the beginning and end of each block. These key instructions commonly include branching commands like *JUMP* and *JUMPI*, as well as *JUMPDEST*, which signals the starting position of a basic block. Both *JUMPI* and *JUMP* instructions read a value from the stack to

[11] Geth. https://github.com/ethereum/go-ethereum.

[12] Ethereum Yellow Paper. https://github.com/ethereum/yellowpaper.

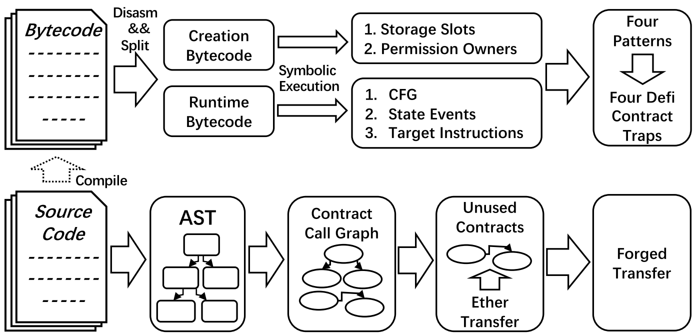


Fig. 1 Overview architecture of *DeFiDefender*

determine the jump destination for the next basic block. Consequently, this establishes an edge between the two blocks within the CFG.

In DeFiDefender, we reused and updated the symbolic execution engine proposed by our previous work named DefectChecker (Chen et al. 2020). The reason why we choose DefectChecker and not others, e.g., Oyente Luu et al. (2016), Securify Tsankov et al. (2018), is that DefectChecker is much faster and more accurate than them Chen et al. (2020). The time to analyze one smart contract by DefectChecker is only 0.15s, while the runtime of Oyente and Securify is 18.48s and 21.55s based on our previous dataset, respectively. Moreover, the primary objective of this paper is to introduce the DCT patterns. Consequently, our aim is not to propose a new symbolic execution engine; rather, we need a tool to aid in detection. Therefore, we reused an existing engine, and the results demonstrate that it effectively achieves our goal.

There are two main updates of DeFiDefender in the usage of the symbolic execution engine of DefectChecker. First, DefectChecker does not support analyzing creation bytecode and contract storage, while all the four DCTs, i.e., TS, SSP, STP, and SP, need to analyze the storage. Thus, DefectChecker cannot be used to analyze them. Second, DefectChecker only supports the Solidity compiler version 0.4.25. However, when deploying smart contracts on Ethereum, developers have the option to select from various Solidity compiler versions, such as v0.4.25 or v0.8.1. These different compiler versions lead to significant variations in the bytecode and AST of the smart contracts.

To address this limitation, we introduced support for opcodes that were previously unrecognized by DeFiDefender. For example, “CREATE2” was introduced at the end of 2019 and was not supported by DefectChecker. This addition allows DeFiDefender to support a wider range of compiler versions. Furthermore, while DefectChecker was limited to utilizing Solidity Compiler v0.4.25 to compile source code into bytecode, DeFiDefender boasts a more flexible approach. It can automatically select from a spectrum of solc versions, supporting up from version 0.4.11 to 0.8.17 (64 compiler versions in total). This adaptability is achieved by integrating 64 Solidity compilers in our tool. In addition, we also optimize the detection rules to be specifically tailored for each Solidity compiler version, e.g., new instructions, swarm hash.[13]

DeFiDefender is optimized for all the versions from v0.4.25 to v0.8.17, which is the latest version at the time this paper is written. The previous dataset used in the DefectChecker was consisted of smart contracts written in lower versions of Solidity(v0.4.25). In order to provide a more realistic representation of real-world smart contracts, the dataset used in the previous study has been updated to include a wide range of higher versions of Solidity. DeFiDefender uses text analysis techniques to match each smart contract with the corresponding compiler for further analysis. Specifically, we employ regular expressions to identify the versions of smart contracts specified in the Solidity code, exemplified by statements such as "pragma solidity 0.4.25". We then systematically adjust the Solidity version, recompiling the contract from the highest to the lowest version until successful compilation is achieved.

4.3 Extracted information

The patterns we used to detect DCTs are based on several pieces of extracted information during the symbolic execution. Below we explain how we obtain them.

(1) ***Creation bytecode and runtime bytecode*** DeFiDefender symbolically executes the creation bytecode and runtime bytecode, respectively. Runtime bytecode is stored on Ethereum directly and can be easily obtained from Etherscan. There are two methods to get the creation bytecode of a smart contract. The first method is synchronizing the whole Ethereum blockchain and storing the creation bytecode during the synchronization. This method needs to instrument the Ethereum clients, e.g., Parity[14] or Geth.[15] Specifically, the first parameter of *opCreate()* in the *core/vm/instructions.go* of Geth contains the creation bytecode. We can add code to store this parameter in our local machine. However, it is time-consuming to synchronize the whole blockchain. The second method is compiling the source code manually by using solc.[16] This method is faster but needs the source code of smart contracts. To prove DeFiDefender can detect the DCTs through bytecode, we choose the first method to get the creation bytecode.

(2) ***Storage slots and contract permission owners*** All the contract storage variables will be stored on the storage persistently when a smart contract is created. A storage variable is a key-value store, and its key is named storage slot. In Ethereum, "*SSTORE* " instruction reads two values from the EVM stack, which represents the storage slot and values respectively. Thus, the storage slots can be identified by "*SSTORE* " on the creation bytecode. A smart contract sometimes needs administrators (also named contract owners) to control the permission of a contract. The administrators' addresses have to be recorded when creating a smart contract, and

[13] Swarm Hash. https://eth.wiki/en/concepts/swarm-hash.

[14] Parity Ethereum Client. https://www.parity.io/ethereum/

[15] Geth. https://github.com/ethereum/go-ethereum

[16] solc,. https://github.com/ethereum/solidity

the addresses will also be stored by instruction "*SSTORE*". An Ethereum address is a 40-bit hexadecimal value and matches the rule of EIP-55 standard,[17] which can be utilized by DeFiDefender to identify the addresses stored on the creation bytecode. In summary, all the storage slots and administrators' addresses are recorded during the execution of the creation bytecode.

(3) ***Control flow graph (CFG)*** A control flow graph (CFG) records all the execution paths of a smart contract. Each node within the CFG represents a single block, which contains a sequence of instructions with no branches in except to the entry and no branches out except at the exit (Chen et al. 2020). While Java bytecode provides explicit jump targets, EVM bytecode necessitates that jump positions be derived from the EVM stack. Through symbolic execution of EVM instructions, we determine the jump positions for each basic block, facilitating CFG construction. Hence, we regard the CFG as a by-product of symbolic execution derived from runtime bytecode.

The CFG (Control Flow Graph) is a graphical tool used in program analysis to represent the control flow structure of a program. It is useful for visualizing program structure, performing program slicing, analyzing data flow, optimizing code, and conducting security analysis. In this paper, the CFG is utilized to capture information about blocks and edges, enabling the quick identification of adjacent blocks when locating critical instructions. This facilitates further detection of whether they conform to the defined patterns without the need for costly symbolic execution.

(4) ***Stack events*** We record all the values in the EVM stack and call them as "Stack Events". Each stack event is a key-value store; its key is the ID of instruction, and its value is the values in the EVM stack when executing this instruction.

(5) ***Target instructions*** To detect DCTs, several target instructions need to be identified during the symbolic execution. The first target instruction is *SLOAD*. *SLOAD* reads one value from the EVM stack, which represents the slot ID of a storage variable. Thus, by detecting this instruction, DeFiDefender can identify whether a function reads the storage. The second target instruction is *CALL*. In Ethereum, the *CALL* instructions are generated by the message calls into an account. For example, calling a function from another contract or library; transferring Ethers between accounts. DeFiDefender aims to identify the *CALL* instruction related to Ether transfer. The *CALL* instruction reads seven values from the EVM stack. The third value represents the transfer amount. Thus, if the value is larger than 0, the *CALL* instruction is related to Ether transfer. We called this kind of *CALL* as a *Money-CALL*. The third target instruction is *SELFDESTRUCT*, which is generated by *selfdestruct* function. This function can transfer all the Ethers on balance to another account and destroy the smart contract.

[17] EIP-55: Mixed-case Checksum Address Encoding. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md.

4.4 Patterns to detect DeFi contract traps

Below we introduce the patterns that we used to detect four DCTs through EVM bytecode and to detect *Forged Transfer* via source code AST analysis. The patterns are presented as several symbolic expressions, and the words in the expression of symbolic patterns are described in Table 2. Besides, we also provide Figures 2, 3 and 4 to help with understanding of the symbolic expressions. Note our tool can be extended with new DCT patterns for both EVM bytecode and AST source code analysis as they are discovered in the future.

$$\begin{gathered}\exists MCALL, ISZERO \in Path_i \wedge \neg Flows(MCALL, ISZERO),\\ \exists Block_h, Block_i, Block_j \in Path_i \rightarrow MCALL \in Block_h, \rightarrow \begin{cases} ISZERO \in Block_i,\\ Storage_s \in Block_j,\\ Block_h > Block_i > Block_j,\\ Storage_s.rewrite == TRUE.\end{cases}\end{gathered} \tag{1}$$

(1) **Tricky send (TS)** Contracts with *TS* have a *address.call.value()()* or *address.send()* in the contracts. These two methods only return a boolean value *false* without affecting the normal running of the contract when Ether transfer fails. A contract with *TS* should satisfy the following conditions, which were shown in Fig. 2 and Eq. (1). *First*, a *Money-CALL (MCALL)* can be found in the $path_i$ ($Block_h$ in the Fig. 2). *Second*, the contract will not check the return value of the CALL instruction, which means the boolean value generated by *MCALL* will not be read by *ISZERO* instruction (There are no flows from *MCALL* to *ISZERO* in Fig. 2).*Third*, the users' balance will be reset. Thus, the subsequent $Block_j$ in the same execution path will change the value of a storage variable $Storage_s$.

$$\begin{gathered}\exists Constr_c(Addr.i == Addr.owner) \in Path_i \mapsto SAT,\\ \exists Block_i, Block_j \in Path_i \rightarrow \begin{cases} Constr_c \in Block_i,\\ Storage_s \in Block_j,\\ Block_i > Block_j,\\ Storage_s.rewrite == TRUE.\end{cases}\end{gathered} \tag{2}$$

(2) **Super Storage Permission (SSP)** Contracts with *SSP* have permission checks for some functions, as the contract should only allow the contract owners to change the storage values. For example, in Listing 1, the modifier *onlyOwner* (L7) is used to ensure only the contract permission owner can change the storage. In Ethereum, all functions of a contract are fused in one stream of instructions (Grech et al. 2019); intra-contract function calls are all realized by jumps (Chen

Table 2 Words in Expression of Symbolic Patterns

Words	Description
$Path_i$	The *i th* path on the CFG
$Storage_s$	A storage variable on Slot ID *s*
Flows(*a*, *b*)	Instruction *a* is poped from the top of the EVM stack, which then is read by the instruction *b*
MCALL	The CALL instruction that used to transfer Ethers
$Block_i$	The *i th* block on the CFG
ISZERO	The ISZERO instruction
Addr.i	The address of *i*
Constr(*expr*)	The constraint of a branch on CFG that checks whether the conditional expression *expr* is satisfying assignment (SAT) or not (UNSAT)
$node_i$	The *i th* node on the CCG

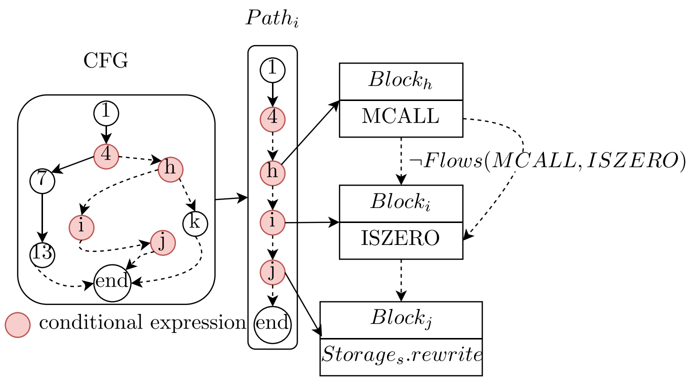


Fig. 2 Tricky Send in DCT

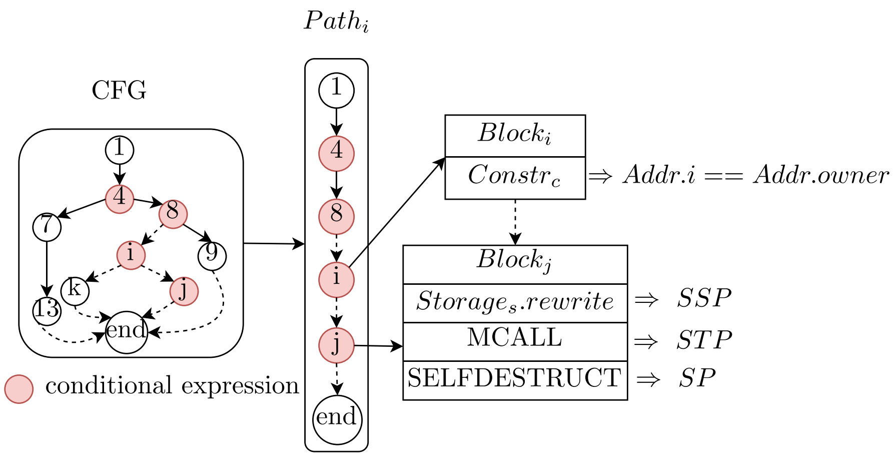


Fig. 3 Super Storage Permission, Super Transfer Permission and Selfdestruct Permission in DCT

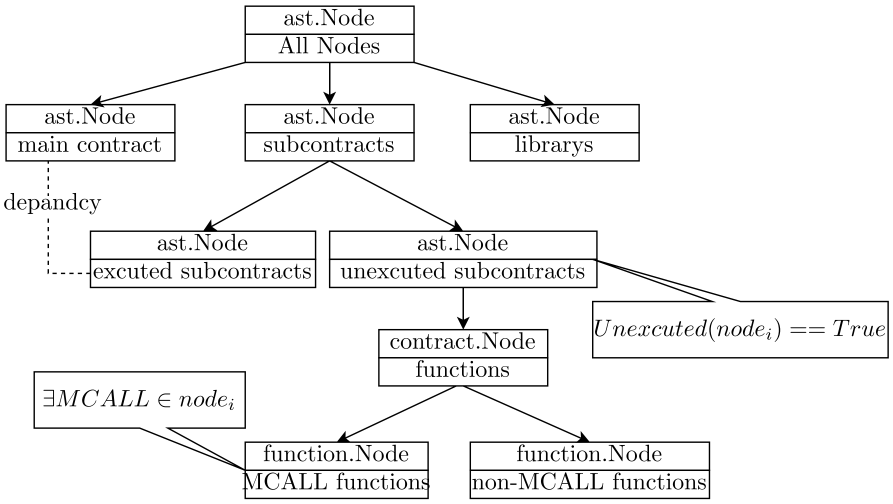


Fig. 4 Forged Transfer Pattern in DCT

et al. 2019)[18] as the method to transfer control. Thus, a path in the CFG can reflect the execution of a function call. DeFiDefender traverses all the paths of the CFG. As shown in Fig. 3 and Eq. (2), for each path, DeFiDefender inspects the conditional expressions. If a constraint of a branch on the CFG checks whether an address *Addr.i* is equal to the contract Owner's address (*Addr.owner*, See Sect. 4.3), it means that a permission check is detected ($Block_i$ checks the constraint in Fig. 3). Then, DeFiDefender identifies whether the storage variables $Storage_s$ are changed in the following block $Block_j$ on the same path with $Block_i$. If so, *SSP* is detected.

$$\exists Constr_c(Addr.i == Addr.owner) \in Path_i \mapsto SAT,$$
$$\exists Block_i, Block_j \in Path_i \rightarrow \begin{cases} Constr_c \in Block_i, \\ MCALL \in Block_j, \\ Block_i > Block_j, \end{cases} \tag{3}$$

(3) **Super transfer permission (STP)** Similarly, misbehaving developers also need to check the permission when transferring Ethers. Thus, as shown in Fig. 3 and Eq. 3, DeFiDefender first traverses all the paths on the CFG and then checks whether an address is equal to the contract owner's address in a constraint $Constr_c$($Block_i$ checks the constraint in Fig. 3). Finally, if Ether transfer(*MCALL* in $Block_j$) is found in the following block in the same path with $Block_i$, an *STP* DCT is detected.

[18] Solidity Document. http://solidity.readthedocs.io

$$\exists Constr_c(Addr.i == Addr.owner) \in Path_i \mapsto SAT,$$

$$\exists Block_i, Block_j \in Path_i \rightarrow \begin{cases} alignedConstr_c \in Block_i, \\ SELFDESTRUCT \in Block_j, \\ Block_i > Block_j, \end{cases} \tag{4}$$

(4) **Selfdestruct permission (SP)** Similar to the previous two traps (as shown in Fig. 3 and Eq. 4), DeFiDefender traverses all of the paths on the CFG and detects the permission check in a constraint $Constr_c(Block_i$ checks the constraint in Fig. 3). Then, if *SELFDESTRUCT* instruction is detected in the following block $Block_j$ in the same path with $Block_i$, a *SP* DCT is detected.

$$\exists MCALL \in node_i \wedge Unexecuted(node_i) == True, \tag{5}$$

(5) **Forged Transfer (FT)** As shown in Fig. 4, DeFiDefender first constructs the abstract syntax tree (AST) of the contract by using the command provided by the solidity compiler.[19] Based on the AST, DeFiDefender builds the contract call graph (CCG). Each node in the CCG can be divided into a main contract, subcontract, or library (The main contract can be obtained from Etherscan directly, the library and the subcontract nodes can be identified by the nodeType field of nodes in AST). Next, DeFiDefender divided the subcontracts nodes into executed subcontracts and unexecuted subcontracts by utilizing a breadth-first algorithm to recursively traverse all dependencies originating from the main contract. By parsing the CCG that starts from the main contract, we can know the executed and unexecuted subcontracts. If there are nodes of unexecuted subcontracts, DeFiDefender will check whether there are Ether transfers related functionalities (*MCALL*), e.g., *address.send(), address.transfer()*. If so, a *FT* is detected.

5 Evaluation

In this section, we aim to conduct a comprehensive evaluation of DeFiDefender by answering the following two research questions (RQs):

RQ1 What is the effectiveness and efficacy of DeFiDefender in detecting DCTs?

RQ2 What is the prevalence of the five defined DCTs in real-world Ethereum smart contracts?

[19] solc,. https://github.com/ethereum/solidity.

5.1 Experimental setup

All experiments were performed on a PC running Mac OS 10.15.6 and equipped with an Intel i7 6-core CPU and 16 GB of memory. Our tool supports the latest compiler version (v0.8.9) of Ethereum smart contracts when the time of writing the paper and is also backward compatible for the older versions, e.g., v0.4.25. We use EVM 1.11.3 to disassemble the bytecode to its opcode.

To evaluate our DeFiDefender tool, we need to obtain real Ethereum smart contract source code. The dataset we used contained 117,926 verified smart contracts uploaded by Feb. 2022. Our dataset has two parts: a small-scale dataset to evaluate the effectiveness and efficacy of DeFiDefender, and a large-scale dataset to evaluate the prevalence of the defined DCTs in the Ethereum. Both datasets only contain DeFi contracts, and non-DeFi contracts are excluded from our evaluation. A key usage scenario of DeFiDefender is where a user wants to check if there are any DCTs on contracts they want to use/invest in. Thus, the users will know that the contract is a DeFi contract even if the source code is not visible, and it is meaningless for users to use DefiDefender to check non-DeFi contracts. To simulate this real usage scenario, we only selected DeFi-related contracts.

5.2 Evaluation methods and metrics

We use eight measurements to evaluate DeFiDefender, i.e., true positive (TP), true negative (TN), false positive (FP), false negative (FN), precision (P), recall (R), F-measure (F), and accuracy. TP and TN indicate DeFiDefender correctly detects that a DCT exists or does not exist in a smart contract, respectively. FP and FN indicate the results that incorrectly predict that a smart contract contains and does not contain a contract defect. Precision, Recall, F-measure, and accuracy can be calculated as:

$$Accuracy = \frac{\#TP + \#TN}{\#TP + \#TN + \#FN + \#FP} \times 100\% \tag{6}$$

$$Precision = \frac{\#TP}{\#TP + \#FP} \times 100\% \tag{7}$$

$$Recall = \frac{\#TP}{\#TP + \#FN} \times 100\% \tag{8}$$

$$F - Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \times 100\% \tag{9}$$

5.3 RQ1:The effectiveness and efficacy of DeFiDefender

Dataset This dataset consists of 700 smart contracts with an average of 514.19 lines of code. To obtain this dataset, we engaged two independent developers, not affiliated with the authors of this paper, who have three years of experience in smart contract development. They first randomly chose 700 smart contracts from all the 117,926 verified Ethereum smart contracts by the time of writing this paper. Then, they manually check whether the contracts are related to decentralized finance. If not, they remove them from the dataset. After that, they randomly choose smart contracts again and identify DeFi-related contracts. This process continued until we got 700 DeFi-related contracts. Next, they read each smart contract carefully and identified whether the contracts contain each of the five defined DCTs. The two developers conducted the manual process independently. Then, they compared their results and discussed any differences until a consensus was reached, to ensure the manual labeled dataset was correct. The process of building the dataset costs around 1.5 months. Table 3 shows the detailed information of this dataset. Among the 700 contracts, 484, 30, 58, 25, 103 of them were compiled with version 0.4+ to 0.8+, respectively.

Result overview Table 4 summarizes the results of applying DeFiDefender to the small-scale dataset. The first column is the DCTs that need to be detected. The second column is the number of DCTs in our dataset. The remaining eight columns are used to evaluate the effectiveness and efficacy of DeFiDefender. Below we discuss the analysis of each DCT and the reasons for the error cases.

1. **Tricky send** There are 19 *Tricky Send* traps in our dataset. DeFiDefender correctly detects 15 of them, but also has 4 false negatives. To construct the CFG, DeFiDefender needs to calculate the jump positions from the EVM stack. However, some jump positions are represented by complicated expressions, and DeFiDefender fails to estimate their values. Thus, some jumps are missing, which leads to the false negative. In some contracts, developers use a variable to store the return value of *address.send()* or *address.call.value()()*. The variable is the input of another function, and the return value of Ether transfer is checked in that function. DeFiDefender fails to identify this situation, which leads to the false positives.

```
if(msg.sender == owner)
    dosomething;
change_storage();
```

Listing 3 False Positive Example for Super Storage Permission

2. **Super storage permission** *Super Storage Permission* is the most frequent contract trap in our dataset. 658 contracts add backdoors to allow their contract owners to change the storage states of smart contracts. DeFiDefender correctly detects 631 of them. It also has 1 false positives and 27 false negatives. To detect this trap, DeFiDefender needs to calculate the storage slot read from the EVM stack. However, the slots are sometimes represented by complicated expressions. For these, DeFiDefender fails to calculate their values and this leads to the false negatives. In some situations, there is a permission check and storage changes in the same path on CFG. However, the permission check does not directly relate to the

Table 3 Some key Information of Smart Contracts in the small dataset

Key information	Numbers
Average lines of code	514.19
Average functions	18.76
Average subcontracts	7.60
Compiler version 0.4+	484
Compiler version 0.5+	30
Compiler version 0.6+	58
Compiler version 0.7+	25
Compiler version 0.8+	103

Table 4 Experimental Results for DeFiDefender

Trap type	# Traps	# TP	# TN	# FP	# FN	A(%)	P (%)	R (%)	F1 (%)
Tricky send	19	15	681	0	4	99.43	100.00	78.95	88.24
Super storage permission	658	631	41	1	27	96.00	99.84	95.90	97.83
Super transfer permission	524	518	170	6	6	98.29	98.85	98.85	98.85
Selfdestruct permission	74	55	626	0	19	97.29	100.00	74.32	85.27
Forged transfer	55	54	645	0	1	99.86	100.00	98.18	99.08
Average	266	254.6	432.6	1.4	11.4	98.17	99.74	89.24	93.85

storage changes. Thus, it leads to the false positives. Listing 3 is an example; the three lines of code are in the same paths on CFG. However, changing storage is not limited by the contract owner.

3. **Super transfer permission** There are 524 *Super Transfer Permission* in our dataset. DeFiDefender correctly identifies 518 of them, with 6 false positives and 6 false negatives. The reasons for these error cases are similar to the *Super Storage Permission*. In addition to these reasons, we find that some contracts use a white list to identify who can transfer Ethers. However, the contract owners can add or remove users in the white list. DeFiDefender fails to identify this situation, which leads to some false negatives.
4. **Selfdestruct permission** We found 74 contracts that contain *Selfdestruct Permission*. DeFiDefender correctly identifies 55 of them, with no false positives and 19 false negatives. DeFiDefender fails to identify some jumps in the CFG. Thus, when traveling the CFG, DeFiDefender misses to identify some cases. There are some functions that have a constraint with two levels of nesting instead of directly checking if it is the contract owner. In such cases, the pattern becomes ineffective. Listing 4 is an example.

```
constructor() public{
    deployer = msg.sender;
}
modifier deployerOnly { require(msg.sender == deployer); _; }
modifier whenInitialized { require(withdrawalAddress != 0x0); _; }
function initializeVestingFor (address account) deployerOnly{
    ...
    withdrawalAddress = account;
    ...
}
function withdrawTokens () private whenInitialized {
    ...
    selfdestruct(withdrawalAddress);

}
```

Listing 4 False Negative Example for SP

5. **Forged transfer** 55 contracts contain *Forged Transfer*. DeFiDefender identifies 54 of them through contract source code analysis, with one false negatives. The reason for this error case is that the independent transfer function which transfer ether do not have the payable modifier, that means the function can not receive or transfer ether. It violates the fundamental principle.Listing 5 is an example.

```
function withdrawEther(uint _amount) public
    onlyEscrow
{
    require(namiMultiSigWallet != 0x0);
    if (address(this).balance > 0) {
        namiMultiSigWallet.transfer(_amount);
    }
}
```

Listing 5 False Negative Example for FT

Answer to RQ1: DeFiDefender is not only highly effective but also highly efficient. DeFiDefender only needs 0.48s to analyze one DeFi smart contract and obtains a high average accuracy (98.17%), precision (99.74%), and recall (89.24%).

5.4 RQ2: The prevalence of DCTs in real-world

Dataset This dataset consists of 20,679 DeFi-related Ethereum smart contracts with an average of 667.5 lines of code, which are extracted from all 117,926 verified contracts. We first tokenized the Ethereum smart contracts into lists of words by punctuation and space that usually do not contain any information. Then, we separate the words according to the rules of Camel Casing.[20] For example, the word "moneyLending" is separated into "money" and "Lending". After that, we converted words into their stemmed form by using Porter's stemmer (Porter et al. 1980). For instance, "lending" is replaced by "lend". Finally, if a smart contract contains one of the finance-related keywords, we regard the contract as a DeFi-related contract. The

[20] Camel Case. https://en.wikipedia.org/wiki/Camel_case/.

finance-related keywords consist of two parts. First, we use the thesaurus provided by Merriam-Webster,[21] which contains 12 synonyms for finances and 75 words related to finances. Then, we add other 7 keywords, i.e., *interest, lend, p2p, loan, credit, reward, bonus*, which are prevalent in the 700 small-scale dataset (dataset used in RQ1) but not included in the Merriam-Webster. We found that all the 700 Ethereum smart contracts used in RQ1 are included in the large-scale dataset. Thus, these 700 smart contracts could be considered as samples of the large-scale dataset, and its evaluation results of RQ1 can also prove the effectiveness and efficacy of our tool in the large-scale dataset.

Result overview In the analysis above, we used a manually labeled small-scale dataset to prove the effectiveness and efficacy of DeFiDefender in detecting defined DCTs. However, this does not show the prevalence of our 5 defined DCTs in real world Ethereum smart contracts. Thus, we ran DeFiDefender on 20,679 DeFi-related Ethereum smart contracts selected from all the verified Ethereum smart contracts on Etherscan. Table 5 shows the frequency of each contract trap on the Ethereum DeFi-related contracts. The second column is the number of contract traps we detected, and the third column is the percentage of the contract traps in our dataset. Notice that DeFiDefender only identifies whether a contract contains a contract trap. Thus, we only count once even if the same kind of contract trap appears several times in the contract.

50.56% and 10.46% of real Ethereum DeFi related Ethereum smart contracts contain Super Storage Permission and Super Transfer Permission, respectively, which are the top two most frequent DCTs in Ethereum. By utilizing these DeFi contract traps, the contract owners can make unscrupulous profit by changing the storage variables or transferring Ethers directly. About 3.67% of DeFi contracts contain *Selfdestruct Permission* trap. The contract owners are able to destroy the contracts and transfer all the balance on the contracts if they want. *Tricky Send* and *Forged Transfer* traps are the least frequent DCTs in Ethereum; only around 1.25% and 0.59% of DeFi contracts contain these traps, respectively.

Besides, we found that only 9900 out of 20,679 (47.87%) Ethereum smart contracts in our dataset do not contain any of the DCTs that we defined in this paper; this means the remaining 10,779 (52.13%) DeFi smart contacts contain at least one DCT. Among these 10,779 contracts, most of them only have exactly one or two DCTs; the numbers are 8077 (39.05%) and 2435 (11.78%), respectively. Only 255 (1.23%) contracts have three DCTs, and 12 (0.06%) contracts have four DCTs. We did not find any contracts that have all the five DCTs introduced in this paper.

Although more than half of the contracts contain DCTs, it does not mean that the owners of these contracts are malicious. For example, contract owners of contracts with *Super Transfer Permission* and *Selfdestruct Permission* have the ability to transfer Ethers that belong to the contract user, but it does not mean they will definitely transfer the Ethers in this way. Some owners add these codes in order to provide a defense against other attacks. Specifically, once bugs are detected by attackers, contracts with these traps give their contract owners the ability to protect

[21] https://www.merriam-webster.com/thesaurus/finances.

Table 5 The Distribution of DCTs in 20,679 Ethereum DeFi related smart contracts

Trap type	# Traps	# Frequency (%)
Tricky send	259	1.25
Super storage permission	10,456	50.56
Super transfer permission	2164	10.46
Selfdestruct permission	759	3.67
Forged transfer	122	0.59

the contract users' assets by transferring Ethers to another account. However, these "privileges" are harmful to the decentralized DeFi ecosystem. On the one hand, it poses "centralization threats" for Ethereum smart contracts and obeys the decentralized intention, which might raise trust concerns. As we introduced before, addressing trust concerns is a key advantage for DeFi compared to traditional financial systems. On the other hand, DCTs are still harmful because they still show that the contract owner has the ability to make unfair profit from the contract, and the users have risks of losing Ethers. The high frequency of contract traps at least shows that the concept of decentralized finance has inherent limitations, as these Ethereum smart contracts cannot ensure the safety of contract users' assets in a zero-trust environment (or without a trusted intermediary) as often claimed.

Answer to RQ2: We observed that only 9900 contracts (47.87%) did not include any of the defined DCTs mentioned in our paper. This implies that the remaining 10,779 contracts (52.13%) belonging to the DeFi category contained at least one DCT. This data underscores the significant prevalence of DCTs in the ecosystem.

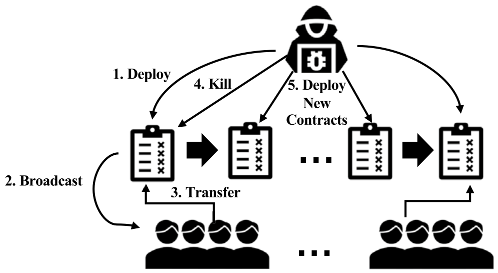
6 Discussion

6.1 Case study

Figure 5 is a real-world example detected by DefiDefender, which shows how misbehaving developers utilize *Selfdestruct Permission* to make unfair profits. First, the attacker[22] deployed a contract named *SmartRoulette*[23] on Ethereum. The *SmartRoulette* contract claims that the users can invest some Ethers—the contract will choose one user as the winner every 256 blocks (about one hour), and the winner can get a certain amount of Ethers as reward. Second, to attract more users take part in the roulette, the attacker develops a website[24] and advertise the website on

[22] Address: 0x6d28515bf27529843f14dc75cc7ee95a4783e3a1.
[23] Address: 0x460f5bf9f5ccfc99243aa4145e4e40c6a6fd9624.
[24] https://smartroulette.io/blockchain.

Fig. 5 An Attack Example of *Selfdestruct Permission*



some platforms/forums, e.g., DApp store.[25] (The screenshots of the two websites are uploaded in case of the shut down.) To gain the trust of users, the attacker publicizes the contract address on their website. In this case, users can check the contract source code and transactions on Etherscan. Also, the attacker transfers 277 Ethers to the contract as a bonus pool to show that the contract has the ability to pay the rewards.

When users check the smart contract on Etherscan, they find the smart contract has enough balance to pay the reward; the contract code is open source; and previous transactions can prove all the winners received the rewards as the contract claimed. The only trap is that the attacker has added a *Selfdestruct* function in the contract. However, this trap is not easy to be found by users through hundreds of lines of code or without enough professional knowledge about DeFi smart contract development.

From contract transactions, we found many users sent a small number of Ethers, e.g., 0.01 Ether, to the contracts for the first time. When they find the contract can work normally, they will send more Ethers to make more rewards. However, when the contract receives a large amount of Ethers in one round, the attackers killed the contract by utilizing the *Selfdestruct* function. All the balance in the contract will then transfer to the attacker's account. After that, the attacker will redeploy the same *SmartRoulette* contract on Ethereum. The newly redeployed contract will obtain a new contract address. The attackers will update the contract address in the website,[4] and use the same way to cheat other victims.

Up to Dec. 2020, the attacker has destructed four *SmartRoulette* contracts. They earned 41 Ethers (about $ 136,300 in Aug, 2021) by utilizing the SP trap in the contract. There are still some live *SmartRoulette* contracts created by the attackers. These contracts also risk being destroyed by attackers in the future.

[25] https://dapp.review/dapp/12327/smartroulette.io.

6.2 Implications

6.2.1 For smart contract developers

It is unavoidable for some contracts to add super-permissions for contract owners, as they need to make profits from the contracts and thus should have abilities to transfer Ethers. However, it might lead to a crisis of trust to the contract users. A good way to balance users' trust concerns and owners' permissions is building a *Decentralized Autonomous Organization* (DAO).[26] In a DAO system, users are given voting rights by using digital tokens to control the DAO, which controls the execution of the contracts. The users who have voting rights (hold tokens) can submit a proposal, e.g., changing the contract storage, executing the selfdestruct function, etc. Then, a group of volunteers called "curators" will check the legality of the submitted proposal and the identity of the submitter. Finally, the users who owned the DAO tokens vote to accept or reject the proposal.

6.2.2 For platform developers

As the example shown in Fig. 5, misbehaving developers might introduce their DApp in their own website or publicize the contract addresses on a DApp store (DappReview). The victim can check the contract code/transactions on Etherscan through the given contract addresses. Usually, the misbehaving developers will allow the contracts to work normally for a period of time to generate some normal transactions. These transactions might mislead users that the contracts are reliable and lead to financial loss when they invest a large amount Ethers. The target users of DeFiDefender are contract users without professional knowledge about smart contracts; although DeFiDefender could warn the risk of DCTs, it might be difficult for users to run DeFiDefender to check the smart contracts before they use them. It would be helpful for the platform developers, e.g., *DappReview*, *Etherscan*, to merge our tool to check whether the smart contracts contain DCTs. If traps are detected, they can add a warning label on the website to inform users of the risks.

It might be helpful for the platform developers, e.g., *DappReview*, *Etherscan*, to merge our tool to check whether the smart contracts contain DCTs. If traps are detected, they can add a warning label on the website to inform users of the risks.

6.3 Importance of DCTs

As we described in Sect. 5.4, although smart contracts with DCTs do not mean that the contracts are definitely malicious, they still show risks for contract users. DCTs act more like "warnings" not definitive smart contract "errors" or "attacks". In software engineering, "warnings" are still important. An analogy is when Android requires apps to show the permissions they use when users install Apps. It is common sense that

[26] Decentralized Autonomous Organization. https://ethereum.org/en/dao/.

social media apps like *WhatsApp* need to read address books, but Android/IOS still requires them to ask for the permissions that they want to use. It is the users' right to know what permissions an app uses. Users will allow permission for some well-respected organizations like WhatsApp, but will pay attention to some non-famous organizations when using their apps. In this paper, we highlight five patterns that inform users of their smart contract risks. They can be considered like the permission "Reading Storage" in Android Apps; users may choose to believe in a well-respected organization but pay more attention to contracts released by non-famous organizations when transferring their money.

6.4 DCTs versus vulnerabilities

As we mentioned in Sect. 1, the key difference between DCTs and vulnerabilities is that vulnerabilities are program "errors" that could lead to financial loss to contract owners and users. In contrast, DCTs are "warnings" that only pose threats to contract users but benefits for contract owners. In some situations, DCT can be an extension and exploitation of a vulnerability by adding additional conditions (code snippets). Thus, some DCTs could be regarded as the "transformation" of vulnerabilities. The "transformation" of vulnerabilities are harmless for the contract owners but allows them to unfairly profit from an unsuspecting contract user.

The *Tricky Send (TS)* and *Selfdestruct Permission (SP)* introduced in this paper could be regarded as the "transformation" of vulnerabilities. Specifically, Luu et al. (2016) introduced a vulnerability named *Mishandled Exception (ME)* which has a similar pattern to the TS DCT. ME only identifies whether the return value of the send method is checked. If not, it could lead to the logic errors of programs, which might cause financial loss for contract owners and users. TS could be considered as a transformation of the ME, as it has one more condition, i.e., a user's balance is cleared after the unchecked send. Thus, the contract owner could benefit from the Ether transfer failure. TeEther Krupp and Rossow (2018) introduced a selfdestruct-related vulnerability (SV) which is similar to our SP DCT. The key difference is that the selfdestruct function could be executed by anyone in the SV, while it could only be executed by contract owners in SP. This difference leads to different consequences, i.e., all the balance on the SV contracts could be stolen by attackers, while only the contract owner can transfer the balance on the SP contracts. The detection methods for SV and SP are also different. A common step to detect SV and SP is generating critical paths which contain the SELFDESTRUCT instruction. The difference is that TeEther checks whether anyone can execute the instruction by checking the constraints, while DeFiDefender has a permission check in the conditional expression to ensure only the contract owners can execute the instruction, which needs an additional analysis for checking the creation bytecode and storage.

6.5 Threats to validity

6.5.1 Internal validity

Since this is the first work that introduces the five DeFi contract traps to the public, there is no dataset for us to evaluate DeFiDefender. Thus, we had to manually label a new dataset, which consists of 700 DeFi-related smart contracts, as our ground truth. The manual labeling process can make the developers of DeFiDefender familiar with the dataset, thus might lead to potential optimization or omissions. To reduce the influence of the dataset, we hired two external collaborators, both with three years of experience in smart contract development. They are required to label the dataset independently and double-check the result. Besides, we have made our dataset publicly accessible via our online repository.

Another key threat is that we use keywords to identify DeFi-related smart contracts in Sect. 5.4. It is possible that some non-DeFi contracts are included, and some DeFi contracts are excluded. Our small-scale dataset introduced in Sect. 5.3 is constructed manually. Thus, we have a higher confidence that these smart contracts are all DeFi-related contracts without any errors. We find that these 700 smart contracts can all be found by the keywords, which shows our method can effectively find DeFi related smart contracts.

6.5.2 External validity

DeFiDefender traverses all the paths on the CFG to detect DCTs. Due to the scalability of Ethereum, it currently cannot support a large-scale project running on the blockchain (Chen et al. 2020). Thus, the average lines of smart contracts in our dataset are about 600 lines, and the biggest contract only has 2239 lines. The current size of smart contracts will not affect the normal running of DeFiDefender. However, in the future, as the supported size of smart contracts increases with the advancement of the Ethereum system, our solution may face a path explosion problem as it traverses all the paths on the CFG.

7 Related work

7.1 Ponzi scheme smart contracts

A Ponzi scheme is a kind of fraud that pays profits to earn from more recent investors to earlier investors.[27] Thus, the Ponzi schemes can make earlier investors believe the products can help make money. Bartoletti et al. (2020) found that many frauds use Ethereum to design Ponzi scheme contracts and claim their products are trustworthy as they are based on the blockchain platform. They manually analyzed 1382 verified smart contracts on Etherscan and found four kinds of Ponzi schemes smart contracts

[27] Ponzi Scheme. https://en.wikipedia.org/wiki/Ponzi_scheme.

according to the patterns used to make money, i.e., array-based pyramid schemes, tree-based pyramid schemes, handover schemes, and waterfall schemes. After that, they expand the collection of Ponzi schemes smart contracts they found by calculating the similarity of bytecode. If the bytecode of a smart contract on the blockchain has a high similarity with the Ponzi scheme contracts they found, the contract is considered as a Ponzi scheme contract. Finally, they found 184 schemes and opened their dataset to the public. Based on the dataset proposed by Weili et al. (2018) design a machine learning-based method to identify whether a contract is a Ponzi scheme without the need for contract source code. They first extracted seven features from contract transactions, e.g., the number of investments and payments. Then, they calculated the frequency of opcodes in smart contracts bytecode. After that, they merged two kinds of features and used XGBoost Chen and Guestrin (2016) to train a model to predict the result.

7.1.1 Differences to our work

Ponzi scheme smart contracts focus on luring victims to pay money to the contract that claims to return high interest. The earlier investors can make profits from more recent investors. These DeFi smart contracts are always fraudulent. In contrast, DCTs focus on how to make money by cheating investors in a different way by adding DCTs to otherwise acceptable DeFi smart contracts. For example, by adding *Super Transfer Permission*, they can transfer the Ethers from the contract balance or change interests by adding *Super Storage Permission*. The DCTs can not only be added to Ponzi scheme smart contracts but also to otherwise acceptable DeFi smart contracts to commit fraud.

7.2 Smart contract vulnerabilities

Previous works (Kalra et al. 2018; Chen et al. 2020a) describe several security vulnerabilities of smart contracts. Among them, a vulnerability named "Unchecked send" reports that some contracts do not check the return value of *address.send()* or *address.call.value()()*, which might lead to errors. Chen et al. (2020b) investigated the reasons why smart contract developers destroy their contracts by using *selfdestruct* function. They found some verified smart contract on Etherscan might include some unused subcontracts, and this kind of contracts were called as "confusing contracts". Since the functions of smart contracts can be called by anyone, they claim that the unused subcontracts might confuse users and reduce the readability of a smart contract. Thus, many developers choose to destroy the contract and deploy a new contract that does not contain unused parts.

Differences to our work The differences, i.e., concepts, detecting methods, have been discussed in Sect. 6.4.

7.3 Honeypot attacks

Torres et al. (2019) introduced honeypot attacks based on Ethereum smart contracts. They found some developers add an obvious flaw in their contracts. The flaw can lead to Ether loss of a smart contract, and attackers can make an unfair profit from it. The attackers need to send a certain amount of Ethers to the flawed contract, and they believe they can drain more Ethers from the contract balance. However, the obvious flaw is usually a trap made by the contract owner. Once attackers send Ethers to the contract, the failure of Ether withdraw makes them realize they only focused on a sole vulnerability that can lead to Ether loss of a contract, but they were aware there is a second vulnerability hidden in the contract. These kinds of traps are called honeypots. Torres et al. (2019) introduced eight kinds of honeypots from three levels, i.e., Ethereum Virtual Machine, Solidity Compiler, and Etherscan Blockchain Explorer. Besides, they developed a tool named HoneyBadger to detect defined honeypots. HoneyBadger consists of three parts, i.e., symbolic analysis, cash flow analysis, and honeypot analysis. They first use symbolic analysis to obtain the CFG. After that, they use the result of symbolic analysis to detect whether the contract is able to receive and transfer Ethers. Finally, HoneyBadger uses predefined patterns to detect honeypot attacks.

7.3.1 Differences to our work

First, honeypots make money from other attackers who have professional knowledge about smart contract programming. The DCTs aim to make an unfair profit from contract users who have no experience in programming. Besides, honeypots are a type of attack, while DCTs are more like a "warning", which makes DCTs more frequent in Ethereum. Specifically, only 460 honeypots are detected from 48,487 smart contracts, while more than half of smart contracts contain DCTs.

8 Conclusion and future work

In this paper, we provide the concept of a DeFi Contract Trap (DCT) to be a code snippet in a DeFi smart contract that can allow unscrupulous contract owners to unfairly profit from unsuspecting contract users. We provide five examples of DCTs and present a symbolic execution based tool, DeFiDefender, to identify such contract traps. Four of them can be detected by DeFiDefender in arbitrary contract bytecode on Ethereum without the need for source code analysis. We use two datasets to evaluate DeFiDefender. The small-scale dataset shows that DeFiDefender is not only highly effective (achieves an average accuracy and F1-score of 98.17% and 93.85% respectively) but also highly efficient (it only needs 0.48 s to analyze one smart contract). Our large-scale dataset analysis shows that the defined five contract traps are prevalent in Ethereum—about 52.13% of DeFi smart contracts contain at least one trap.

Our analysis currently focuses only on determining whether a smart contract contains specific DCT patterns, without delving into the intentions of the contract owners. In the future, we plan to assess the historical transactions associated with the contract. This will allow us to determine whether the contract owner has engaged in malicious activities to users.

Author contributions Jiachi Chen wrote the main manuscript text for the paper. Jiachi Chen and Jianghu designed the tool together. Xin Xia proposed the main idea of the paper. Xin Xia and David Lo and John Grundy helped polish the paper and discuss the idea with Jiachi Chen Zhipeng Gao contributed to the dataset of the paper. Ting Chen provided some important ideas and helped polish the paper.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

References

Bartoletti, M., Carta, S., Cimoli, T., Saia, R.: Dissecting ponzi schemes on ethereum: identification, analysis, and impact. Futur. Gener. Comput. Syst. **102**, 259–277 (2020)

Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd Acm Sigkdd International Conference on Knowledge Discovery and Data Mining, pp. 785–794. ACM (2016)

Chen, T., Li, Z., Zhang, Y., Luo, X., Chen, A., Yang, K., Hu, B., Zhu, T., Deng, S., Hu, T.: Dataether: data exploration framework for Ethereum. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1369–1380. IEEE (2019)

Chen, T., Li, Z., Zhang, Y., Luo, X., Wang, T., Hu, T., Xiao, X., Wang, D., Huang, J., Zhang, X.: A large-scale empirical study on control flow identification of smart contracts. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11. (2019). IEEE

Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., Chen, T.: DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. arXiv:2009.02663 (2020)

Chen, J., Xia, X., Lo, D., Grundy, J.: Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. arXiv preprint arXiv:2005.07908 (2020)

Chen, T., Zhang, Y., Li, Z., Luo, X., Wang, T., Cao, R., Xiao, X., Zhang, X.: 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)

Chen, T., Zhu, Y., Li, Z., Chen, J., Li, X., Luo, X., Lin, X., Zhange, X.: Understanding Ethereum via graph analysis. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications, pp. 1484–1492. IEEE (2018)

Chen, Y., Bellavitis, C.: Blockchain disruption and decentralized finance: the rise of decentralized business models. J. Bus. Ventur. Insights **13**, 00151 (2020)

Chen, T., Li, Z., Zhu, Y., Chen, J., Luo, X., Lui, J.C.-S., Lin, X., Zhang, X.: Understanding ethereum via graph analysis. ACM Tran. Intern. Technol. (TOIT) **20**(2), 1–32 (2020)

Chen, J., Xia, X., Lo, D., Grundy, J., Yang, X.: Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges. Empir. Softw. Eng. **26**(6), 117 (2020)

Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., Chen, T.: Defining smart contract defects on ethereum. IEEE Trans. Softw. Eng. **48**(1), 327–345 (2020)

Ethereum Foundation: Ethereum's White Paper. https://github.com/ethereum/wiki/wiki/White-Paper (2014)

Grech, N., Brent, L., Scholz, B., Smaragdakis, Y.: Gigahorse: thorough, declarative decompilation of smart contracts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1176–1186. IEEE (2019)

Gudgeon, L., Werner, S., Perez, D., Knottenbelt, W.J.: Defi protocols for loanable funds: interest rates, liquidity and market efficiency. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies, pp. 92–112. (2020)

Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium (NDSS'18) (2018)

Krupp, J., Rossow, C.: {teEther}: Gnawing at ethereum to automatically exploit smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 1317–1333. (2018)

Lin, L.X., Budish, E., Cong, L.W., He, Z., Bergquist, J.H., Panesir, M.S., Kelly, J., Lauer, M., Prinster, R., Zhang, S., et al.: Deconstructing decentralized exchanges. Stand. J. Blockchain Law Policy **2**, 8 (2019)

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269. ACM (2016)

Popescu, A.-D.: Decentralized finance (defi)-the lego of finance. Soc. Sci. Educ. Res. Rev. **7**(1), 321–349 (2020)

Porter, M.F.: An algorithm for suffix stripping. Program **14**(3), 130–137 (1980)

Schär, F.: Decentralized Finance: On Blockchain-and Smart Contract-based Financial Markets. Available at SSRN 3571335 (2020)

Szabo, N.: Formalizing and securing relationships on public networks. First Monday (1997)

Torres, C.F., Steichen, M.: The art of the scam: demystifying honeypots in ethereum smart contracts. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 1591–1607. (2019)

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82. (2018)

Wang, Q., Li, R., Wang, Q., Chen, S.: Non-fungible token (nft): Overview, evaluation, opportunities and challenges. arXiv preprint arXiv:2105.07447 (2021)

Weili, C., Zibin, Z., Jiahui, C., Edith, N., Peilin, Z., Yuren, Z.: Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology. In: Proceedings of the 2018 World Wide Web Conference on World Wide Web, pp. 1409–1418. International World Wide Web Conferences Steering Committee (2018)

Werner, S.M., Perez, D., Gudgeon, L., Klages-Mundt, A., Harz, D., Knottenbelt, W.J.: Sok: Decentralized finance (defi). arXiv preprint arXiv:2101.08778 (2021)

Authors and Affiliations

Jiachi Chen[1] · Jiang Hu[1] · Xin Xia[2] · David Lo[3] · John Grundy[4] · Zhipeng Gao[5] · Ting Chen[6]

✉ Xin Xia
xin.xia@acm.org

Jiachi Chen
chenjch86@mail.sysu.edu.cn

Jiang Hu
hujiang5@mail2.sysu.edu.cn

David Lo
davidlo@smu.edu.sg

John Grundy
John.Grundy@monash.edu

Zhipeng Gao
zhipeng.gao@zju.edu.cn

Ting Chen
brokendragon@uestc.edu.cn

[1] The School of Software Engineering, Sun Yat-sen University, Zhuhai, China

[2] The Software Engineering Application Technology Lab, Huawei, China

[3] The School of Information Systems, Singapore Management University, Singapore, Singapore

[4] The Faculty of Information Technology, Monash University, Melbourne, Australia

[5] University of Electronic Science and Technology of China, Chengdu, China

[6] Shanghai Institute for Advanced Study, Zhejiang University, Shanghai, China