

DEPOSafe: Demystifying the Fake Deposit Vulnerability in Ethereum Smart Contracts

Ru Ji^{1*}, Ningyu He^{2*}, Lei Wu³, Haoyu Wang¹, Guangdong Bai⁴, Yao Guo²

¹ Beijing University of Posts and Telecommunications, Beijing, China

² Peking University, Beijing, China ³ Zhejiang University, Hangzhou, China

⁴ The University of Queensland, Australia

Abstract—Cryptocurrency has seen an explosive growth in recent years, thanks to the evolvement of blockchain technology and the economic ecosystem. Besides Bitcoin, thousands of cryptocurrencies have been distributed on blockchains, and hundreds of cryptocurrency exchanges are emerging to facilitate the trading of digital assets. However, it also attracts the attentions of attackers. *Fake deposit*, as one of the most representative attacks (vulnerabilities) related to *exchanges* and *tokens*, has been observed in the blockchain ecosystem from time to time, causing a large amount of financial loss. However, besides a few security reports, our community lacks of the understanding of this vulnerability, and we are still unaware of the *scale* and the *impacts* of this vulnerability. In this paper, we take the first step to demystify the fake deposit vulnerability. Based on the essential patterns summarized, we have implemented DEPOSafe, an automated tool to detect and verify (exploit) the fake deposit vulnerability in ERC-20 smart contracts. DEPOSafe relies on several key techniques including *symbolic execution based static analysis* and *behavior modeling based dynamic verification*. By applying DEPOSafe to 176 thousands of ERC-20 smart contracts, we have identified over 7,000 vulnerable contracts that may suffer from two types of attacks. Our findings demonstrate the urgency to identify and prevent the fake deposit vulnerability.

I. INTRODUCTION

As the first generation blockchain platform, Bitcoin [1] demonstrated that it is possible to use the Internet to make a decentralized value-transfer system that can be shared across the world and virtually free to use. Due to the limitation of performance and scalability issues, however, it is difficult (if not impossible) for Bitcoin to support complex applications. To this end, Ethereum [2] was proposed to allow users to create *DApps* (Decentralized Applications) by developing *smart contracts*, which have been regarded as the most creative blockchain technique after Bitcoin.

Up to April 2020, there exists more than 25 million smart contracts in Ethereum. However, only 0.36% of them have opened up the source code according to our dataset, which reflects the contradiction between security and privacy. Besides, Ethereum uses *Ether* as its official cryptocurrency, which can be held and transferred between *accounts* in Ethereum network. Except for *Ether*, Ethereum also allows users to create and issue cryptocurrencies (*tokens*). To standardize the behaviors of tokens, Ethereum proposes a technical standard named ERC-20 [3] to ensure the interoperability between tokens. As a result, Ethereum has attracted lots of attentions

and become one of the most representative second generation blockchain platforms. By the end of April 2020, there are over 176K ERC-20 tokens on the Ethereum platform.

With the rapid development of Ethereum, *cryptocurrency exchange* (*exchange* for short), as the third-party supporting service, has emerged to support *fiat-to-crypto* and *crypto-to-crypto* trades. According to their trading models, exchanges can be categorized into two types, *centralized exchange* (CEX) and *decentralized exchange* (DEX). CEX, as the name suggests, requires a central entity as the intermediary to finish the tokens transfer between its users. Therefore, the trustworthiness of the middle man plays an important role in this transaction model. Contrary to CEX, DEX removes the entity in the middle to store data, and finishes the token exchange by a matchmaking tradeoff model [4]. Theoretically, a DEX is composed of smart contract(s) without the need of human intervention. Therefore, DEX has been developed to allow peer-to-peer trading of cryptocurrencies without an intermediary but solely depends on smart contracts. Such a trading model can not only guarantee the privacy of users¹, but also ensure the behaviors of transaction strictly follow the logic encoded in smart contracts.

Unfortunately, a number of security issues were found in tokens and exchanges, which attracted attentions of attackers as well. First, a number of attacks targeting exchanges have been observed in the wild [6]–[8], which caused great financial losses. Besides, the ERC-20 related bugs have been underestimated by developers [9]. Namely, a token may accidentally be created as non-standard without fully understanding the ERC-20 standard. As a result, **the combination of the flawed verification of exchanges and non-standard implementation of tokens might cause severe damages.**

Fake deposit is one of the most representative vulnerabilities related to both exchanges and tokens. It is publicly well known due to the recent security reports and news [10]–[13]. The attacks exploiting this vulnerability have remained active for a long time, observed in Ethereum and other platforms (e.g., USDT and EOSIO). Unlike other vulnerabilities, the fake deposit vulnerability is conditioned by two requirements, i.e., the non-standard implementation of a token and the flawed verification of an exchange respectively. To be specific,

¹Although it is consistent with the anonymity of blockchain, the abandon of Know Your Customer (KYC) [5] policy is still a concern.

*Equal contribution of the first two authors.

deposit means a user transfers a certain type of token into the exchange. A malicious user, however, could take advantage of flaws of smart contracts of that token and deficient verification mechanism of exchange to achieve a *fake deposit*, in which the amount transferred is unaffordable for the attacker. Consequently, a malicious user can deceive the exchange and gain huge profit with nearly no cost. To date, the details and impact of this vulnerability have not been well studied.

This Paper. We take the first step to demystify the *fake deposit* vulnerability in Ethereum and measure the potential security impacts by analyzing a large number of Ethereum ERC-20 smart contracts. We have analyzed over 176,000 ERC-20 contracts deployed between July 2015 to March 2020. Specifically, we first characterized the vulnerability and figured out the essential patterns. Accordingly, we implemented DEPOSafe, an automated tool to identify and verify (exploit) the *fake deposit* vulnerability. DEPOSafe consists of two major components. The first component is a static detector implemented based on *mythril* [14], an open-source symbolic execution engine towards Ethereum to efficiently perform the detection. After that, to eliminate the false positives introduced by the static analysis, we built a dynamic validator to emulate the behaviors between token and exchange when users deposit tokens into exchange, in order to confirm the existence of the *fake deposit* vulnerability. To facilitate our analysis, we categorized these attacks into two types according to the exchanges, i.e., attacks targeting CEX as *Type-I* and attacks targeting DEX as *Type-II*, respectively. After applying DEPOSafe to over 176,000 ERC-20 smart contracts we collected, we have identified 56 and 7,716 smart contracts that could be exploited by Type-I and Type-II attacks respectively. The technical details of the flawed implementation of token and exchange are presented in §III; the attack emulations for both types of attacks are depicted in §IV.

Contributions In summary, this paper makes the following main research contributions.

- We proposed DEPOSafe, the first framework aims to 1) identify if the smart contract is compliant with the ERC-20 standard rules and subject to fake deposit vulnerability; 2) automatically exploit the vulnerable ERC-20 smart contract and prove the loophole’s existence.
- By applying DEPOSafe to over 176K ERC-20 smart contracts, we have identified more than 7,000 ERC-20 smart contracts with fake deposit vulnerability, which can be exploited by the Type-I or Type-II attacks. Our investigation shows that the top 10 most influential but vulnerable ERC-20 tokens account for 984 million USD capitalization; and the affected DEXes still have millions USD volumes per day.
- We reported various findings based on the analysis of collected data and further propose mitigation of this vulnerability. Moreover, the vulnerability dataset will be released to the community for further study.

II. BACKGROUND

In this section, we will briefly introduce the necessary background knowledge of Ethereum [2] to facilitate the understanding of our further analyses.

A. Ethereum Account and Transaction

In Ethereum, account is the basic unit to identify an entity in the network. An account is identified by a fixed-length hash-like address. Additionally, the account in Ethereum can be divided into two types: *External Owned Account* (EOA) and *Contract Account*. An EOA is an ordinary account who can transfer tokens, invoke deployed smart contracts and store received tokens. Moreover, an EOA can deploy a smart contract into a Contract Account². Specifically, an account can deploy a smart contract by sending a transaction that contain the bytecode of smart contract to address 0x0. After that, all accounts in the Ethereum network are able to invoke the smart contract residing in the Contract Account.

Accounts or smart contracts can interact with each other by sending a *transaction*, which consists of a bunch of data that will be parsed and executed by the target smart contract. To be specific, the data consists of the signature of designated function and its required arguments. Notice that, to minimize the size of the transaction transferred, Ethereum matches functions by its first four bytes of the signature that is calculated by the Keccak256 hash function [15]. For example, the function `transfer(address, uint256)` will be identified by 0xa9059cbb. Once the target address successfully handles the received transaction, the transaction will be recorded online soon and cannot be erased or modified. However, if something goes wrong within executing a transaction, all of the modifications resulted from this transaction will be reverted.

B. Ethereum Virtual Machine (EVM)

EVM is a simple stack-based architecture, and the function of the EVM stack is to store the results of intermittent execution of bytecode instructions (*opcodes*). All the operands of opcodes and the calculation result are pushed onto the *stack*, a basic data structure in EVM. In a nutshell, EVM can provide a runtime environment for smart contracts that have been compiled to bytecode to be executed, manage execution of the transaction and transit the blockchain to its new state.

Except for stack, data in EVM can also be stored in other areas: *memory*, *storage* and *calldata*. To be specific, storage is a key-value mapping that persists between function calls and transactions. As the data in storage are recorded on the blockchain, it charges more to create and update entities in storage. Moreover, EVM will assign a *slot ID* for each variable stored in storage to identify it. The slot ID is determined at compilation time and strictly based on the variables order in the contract code. In contrast, retrieving and inserting data from memory is much more cheaper. The memory area, however, will be erased after the current transaction. Besides above areas, *calldata* is used to store external calls to functions, and it is a read-only byte-addressable space.

²A smart contract can also deploy another smart contract.

C. Smart Contract and Bytecode

A smart contract is a collection of code and data that reside in Contract Account. They are typically written in high level languages, such as Solidity [16]. They are used to implement arbitrary rules as well as guarantee to produce the same result for decentralized parties. Smart contracts exist and execute in bytecode format, which is compiled from source code.

The Ethereum bytecode is made up of 144 opcodes [17]. Additionally, each opcode is encoded as one byte, and represented in hexadecimal format. For example, `SSTORE` is encoded as `0x55` in EVM. Opcode takes zero or multiple arguments to achieve its functionality. As we mentioned in §I, only 0.36% of contracts have opened up the source code. In order to cover all these contracts, we decide to implement DEPOSafe as a bytecode-level analyzer (see Section IV).

Additionally, the bytecode of smart contract is composed of three parts: *creation code*, *runtime code*, and *swarm code*. Creation code includes constructor logic and constructor parameters of a contract. It is generated when the bytecode is compiled and it will be executed only once at the time of deployment. Runtime code will eventually be stored on-chain. It details the logic and behavior of each function in contract. However, it does not contain any of the constructor parameters. Swarm code is a little bit different. It does not have any practical meaning and can not be executed by EVM. The swarm code is only a string of hash that is calculated by metadata of current smart contract to index it in database.

```

1 function transfer(address to, uint tokens)
  public returns (bool success) {
2   balances[msg.sender] = safeSub(balances[msg.
    sender], tokens);
3   balances[to] = safeAdd(balances[to], tokens);
4   emit Transfer(msg.sender, to, tokens);
5   return true; }
6
7 function transferFrom(address from, address to
  , uint tokens) public returns (bool
    success) {
8   // update balances and allowances
9   emit Transfer(from, to, tokens);
10  return true; }

```

Listing 1. Standard `transfer` and `transferFrom` in ERC-20 token

D. ERC-20 Standard

As one of the most popular and well-known technical standards in Ethereum, *ERC-20* specifies six mandatory functions (`totalSupply`, `balanceOf`, `transfer`, `transferFrom`, `approve`, and `allowance`) for the benefit of Ethereum developers. As shown in listing 1, the `transfer` uses function `safeAdd` and `safeSub` to update the balance table. If the invoker does not have enough token, they will throw exceptions and terminate the current transaction. As for `transferFrom`, except for the update of balance table, it also checks if the caller has enough allowance to transfer such an amount of tokens. Any of these three verification fails, the transfer will be terminated immediately.

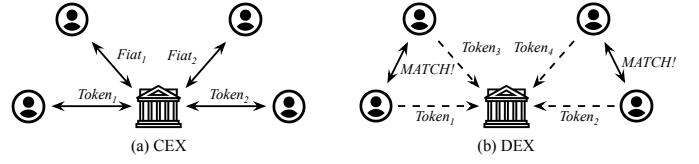


Fig. 1. The trading model of CEX and DEX respectively.

E. Exchange and Deposit

1) *CEX*: CEX is an intermediary between users in trading process (see Fig. 1(a)). Typically, the centralized entity is managed by a trustworthy organization or company. However, once the centralized server is down or compromised, it is possible for data breach or even financial loss for both exchange and users. Nevertheless, CEX has its advantages: 1) it has ability to achieve quick transactions and support multiple users at once; 2) it supports the exchange between fiat and tokens, or even tokens from different platforms; 3) the trading model determines its scalability and quick response against attacks.

Depositing tokens into CEX is simple. The user invokes a transaction with certain amount of tokens to a designated address. After that, the CEX server will verify if the transaction is successful and update the balance of the user in its database.

2) *DEX*: As depicted in Fig. 1(b), DEX, unlike CEX such as Coinbase [18] and Binance [19], has no central entity which is managed by a specific company or a person focusing on making a profit. The DEXes put the control of funds and trading back to users. In other words, DEX does not store user assets, so neither hacker attacks nor the total collapse of the DEX can lead to a loss of funds. To compete with the traditional CEXes and attract users, these DEXes adopt various business logic. For example, in EtherDelta [20], maker and taker agree on trading off-chain and execute it on-chain, while IDEX [21] provides real-time order book updates, which is live on the Ethereum MainNet. Despite these diverse business logic, DEXes share similar mechanisms, protocols and data models, i.e., the actual trade is executed through a smart contract on blockchain. Therefore, once the transaction is confirmed by the miners, it is impossible to withdraw or cancel it due to the irreversibility. Additionally, all the behaviors through DEX strictly follow the code of its smart contract.

```

1 function depositToken(address token, uint
  amount) {
2   if (msg.value>0 || token==0) throw;
3   if (!Token(token).transferFrom(msg.sender,
    this, amount)) throw;
4   tokens[token][msg.sender] = safeAdd(tokens[
    token][msg.sender], amount);
5   Deposit(token, msg.sender, amount, tokens[
    token][msg.sender]);
6 }

```

Listing 2. `depositToken` function

To trade tokens through DEX, users have to deposit tokens into smart contract of DEX in advance. Therefore, the user

has to firstly grant the permission to DEX to make it eligible; then the user will call the `depositToken` function which is detailed in listing 2. As we can see, the `depositToken` takes the token address `token` and deposit value `amount` as arguments, then calls the `transferFrom` in address `token` to transfer money to DEX’s address.

III. FAKE DEPOSIT VULNERABILITY

As aforementioned, the attack against *fake deposit vulnerability* depends on both entities of tokens and exchanges. Therefore, in this section, we will detail the incorrect implementation in ERC-20 token’s smart contracts, and also the two types of deficient verification of exchanges. The combination of these two conditions leads to the fake deposit vulnerability.

A. Non-standard Implementation of Tokens

Though ERC-20 enforces the implementation of these interfaces (see §II-D), the standard does not specify the implementation details, an incorrect or improper implementation could lead to vulnerable contracts. To be specific, the official guideline recommends developers to throw an exception if there are insufficient tokens in the caller’s balance to spend, as shown at line 2-3 in listing 1. The `safeAdd` and `safeSub` would throw an exception if the overflow happens.

```

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

```

Listing 3. An example of a vulnerable implementation of `transfer`.

However, some developers use a conditional statement to check the caller’s balance instead of the assertion statement (see listing 3). If the balance of the caller (identified by `msg.sender`) is insufficient, the transfer will return false at line 7. Unfortunately, no matter which value returned, the current transaction will not be terminated. *This gap between the actual behavior and the developer’s expectation breaks the guideline and leads to the vulnerability.*

B. Flawed Verification of Exchanges

Except for the vulnerabilities that reside in the ERC-20 token smart contract, successfully exploiting the fake deposit also relies on the implementation flaws in exchanges. Therefore, we then introduce two types of deficiencies which lead to the security threats of exchanges.

1) *Flawed Token Verification of DEXs*: After a thorough manual inspection towards current (and past) mainstream DEXes, we found that most of them perform deposit logic by calling the function `depositToken` which is detailed in listing 2. As line 3 shows, the DEX invokes the traded token’s `transferFrom` (or sometimes `transfer`) to perform deposit. Notice that the DEX is responsible for auditing the token’s smart contract that is traded on its platform to ensure financial security. However, imagine a token’s `transferFrom` is encapsulated by `safeTransferFrom`, where it performs security check and then invokes the real `transferFrom`. If the DEX does not audit this smart contract or neglects this unsafe implementation, there seems no security check when DEX calls `transferFrom` at line 3.

2) *Flawed Back-end Verification of CEXs*: When user deposits tokens into CEXes, the depositing amount is parsed and determined by CEX’s back-end server. However, a deficient verification strategy may result in unexpected behaviors. To be specific, some CEXes do only verify the `status` field and `_value` in the input field of a transaction, which is reported by security analysts [22]. The `status` indicates whether the current transaction is terminated with nothing unusual; the `_value` is the value of arguments in the `transfer` function. Therefore, if a transaction is terminated by “return False;” instead of throwing an exception, the `status` code will still be 1, which means current transaction is terminated normally. Unfortunately, some CEXes rely on such an insufficient verification to determine how many tokens are deposited.

IV. DEPOSafe

To evaluate if a smart contract is vulnerable to the fake deposit attack, we implemented DEPOSafe. Fig. 2 shows the overall work process of DEPOSafe. It takes the runtime bytecode of smart contracts with their corresponding addresses as input, and generates a security report through a pipeline composed of two parts: *static detector* and *dynamic validator*. Specifically, the static detector is implemented based on *mythril* [14], in which we screen out the addresses which may be vulnerable to the fake deposit loophole. Due to the inherent false positives of static analysis, we further implemented in DEPOSafe a dynamic validator to verify the flagged smart contracts are indeed vulnerable to fake deposit. The dynamic validator takes advantage of *web3* [23], which is a collection of libraries for interacting with a local or remote Ethereum node. We used it to interact with our local private chain provided by *ganache-cli* [24], which is a customizable blockchain emulator. We detail both components in the following.

A. Static Detector

To efficiently detect the fake deposit vulnerability, we propose a static detector relying on symbolic execution to perform the detection. The *mythril*, which we relied on, implements an EVM-like virtual machine, ~~thus so it has the ability is able~~ to imitate the behaviors of a given smart contract on bytecode level. Therefore, our detector firstly symbolic executes the bytecode to traverse all the feasible paths. Meanwhile, it

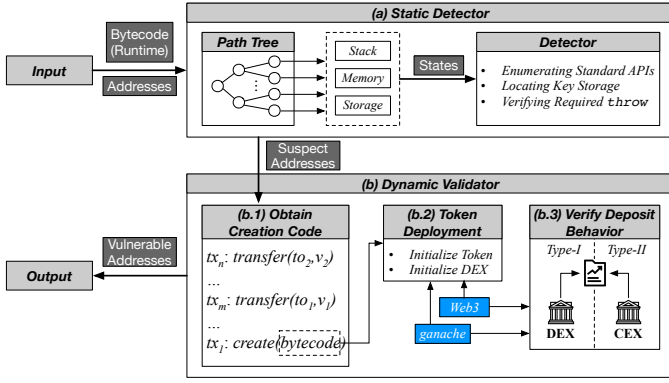


Fig. 2. An overview of DEPOSafe.

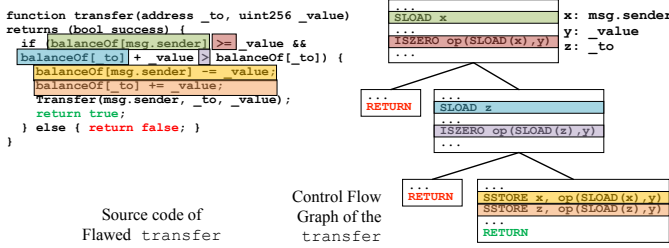


Fig. 3. The detection logic of static detector towards a flawed transfer.

records the virtual machine state (including stack, memory, storage, etc.) after each instruction. After that, based on ~~these generated information~~ the recorded information, the detector is capable of ~~detecting the existence of this vulnerability~~ identifying vulnerabilities.

As ~~the details discussed~~ discussed in §III-A, to accurately detect the incorrect implementation of transfer and transferFrom, we divide the whole static analysis into three steps, including *enumerating standard APIs*, *locating key storage*, and *verifying required throw*. Specifically, in the first step, we use the signature of functions to find out if the smart contract implements any of the six standard functions; in the second step, we use the opcode SLOAD and SSTORE to pinpoint the storage slots of ~~these those~~ key arguments, i.e., balanceOf and allowance; in the last step, we check the final opcode of relevant paths to identify whether the exception is thrown ~~away to meet~~ as required by the ERC-20 standard. The detection procedure against the flawed transfer listed in listing 3 is shown in Fig. 3 and details of these three steps are shown below:

1) *Enumerate Standard APIs*: As the existence of transfer and transferFrom is the prerequisite of the fake deposit vulnerability, so firstly we ~~will~~ check if the smart contract is compliant with the standard ERC20 token interface [3]. In other words, the signatures of those six standard functions should be declared in the smart contract’s bytecode. As we can see from Fig. 3, after ~~we identify~~ identifying the existence of transfer (including transferFrom but we use transfer as an example here), we will generate the Control Flow Graph (CFG) and symbolic execute each paths.

2) *Locate Key Storage*: If the transfer is implemented, we ~~will~~ then verify ~~if whether~~ it follows the official standard, i.e., ~~throw~~ an exception if ~~the message caller’s account balance does not have enough tokens to spend~~ there is insufficient tokens in the caller’s account. Similarly, we also examine the implementation of balanceOf and allowance in transferFrom as well.

Take function transfer as an example, ~~w~~. We have to pinpoint the storage slots used by balanceOf firstly. As the mythril records the virtual machine state when it executes the opcodes of the smart contract, we extract the recorded states of function transfer and traverse all of them to identify the addresses from which the balanceOf is SLOAD (as the balance table is stored in storage (see §II-B) and has to be retrieved by instruction SLOAD in EVM). As we can see from Fig. 3, the green block indicates where it retrieves the balance of msg.sender, i.e., the caller. We mark the label x as *storage address*. Then, we traverse the storage addresses to examine whether any of them is treated as target address by SSTORE (like SLOAD, EVM has to update variable in storage by instruction SSTORE) in transfer. We can see the yellow block which indicates the balance of msg.sender is updated and stored in storage with label x. Therefore, the balanceOf in transfer is updated, which could lead to the actual change of balance data.

3) *Verify Required throw*: Similarly, the success of second step is the precondition of this step. In this step, we detect if the balanceOf is protected and properly handled. Mythril generates all the feasible paths of the transfer function (if it exists), ~~thusso~~ we ~~will~~ screen out all the instructions that indicates a normal termination of current path, i.e., STOP and RETURN, which means these paths are not ~~thrown away~~ ~~baigd: a bit weird to say path is thrown away~~ by assertion. Like the RETURN colored by green and red in Fig. 3, they all represent the termination of that path.

Then, we will traverse these paths backward to ~~find search~~ for the instruction ISZERO, which is used at the comparison between values. For example, from any of the RETURN opcode in Fig. 3, we can find a ISZERO backward. As for the two arguments compared by ISZERO, if one of them is the target address of SSTORE, it means the balanceOf is protected by if-else before updating. Hence, we treat this ISZERO as a *protected node*, as the red block and purple block in Fig. 3. For all the paths forking from the protected nodes, if none of them is terminated by revert instructions, i.e., REVERT and ASSERT_FAIL, we can ensure that this smart contract ~~not-follows~~ ~~does not follow~~ the official guideline. In other words, neither of branches forked by if-else is terminated by assertion and ~~thrown away~~, such that the smart contract may be ~~vulnerable~~ subject to fake deposit vulnerability.

Though we proposed a well-designed strategy to identify the fake deposit vulnerability in ERC-20 smart contracts, we have to face the introduced false positive which is the inherent problem for static analysis method. For example, as shown in listing 1, some transfer implements security check by

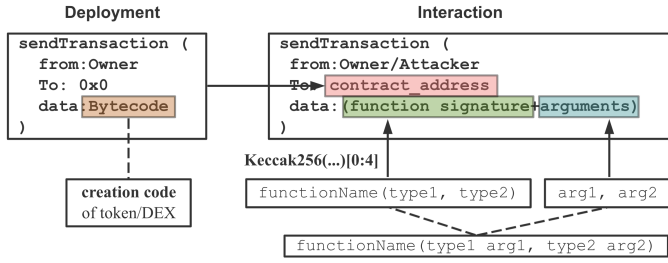


Fig. 4. The deployment and interaction between smart contracts in favor of web3.

safeAdd or safeSub. Such a way, however, is possible to import false positive. Notice that, we will comprehensively analyzing all the false positive cases in §V-B. Consequently, we implement a dynamic validator after static detector because of two reasons: 1) eliminating the false positive introduced by static analysis and increasing the precision; 2) achieving automatically EXP generation towards fake deposit to validate the existence of vulnerability.

B. Dynamic Validator

To distinguish the true and false positive cases generated by the preceding component, we decided to mimic the deposit behavior between tokens and exchanges. Therefore, in this component, we divide the whole validation process into three steps: *obtaining creation code*, *test environment initialization*, and *verifying deposit behaviors*.

1) *Obtain Creation Code*: As the static detector only passes the addresses of those token smart contracts that may be vulnerable to fake deposit behaviors, we have to firstly obtain the creation code, which indicates which contract is indeed deployed and the corresponding initial value of parameters (see §II-C). To be specific, we firstly implement a crawler to grab all the transactions invoked and received from etherscan.io [25] (a well-known and credible browser for Ethereum) according to the addresses. Then, we only keep the oldest one, which is the contract deployment transaction (see §II-A), to parse. Finally, we can obtain the creation code and the initial values from input data field.

2) *Test Environment Initialization* *Lei: what is the meaning here?*: After we got the creation code of tokens, we have to deploy them on our private chain to perform the following tests to avoid ethical and financial problem. To be specific, we set up a private chain and interact with the deployed contract in favor of ganache and web3 respectively. Moreover, as shown in Fig. 4, the `sendTransactions` provided by web3 enable the contract deployment and function invocation from us.

Except for tokens, as DEXes also perform all their functionalities in smart contracts (see §II-E), we also deployed the DEXes in our test environment.

3) *Verify Deposit Behaviors*: As shown in Fig. 4, to invoke the functions in deployed smart contracts, we achieve that goal by `sendTransaction`. To be specific, we can indicate the invoker, receiver, and the specific function and its parameters in `from`, `to`, and `data` respectively in `sendTransaction`.

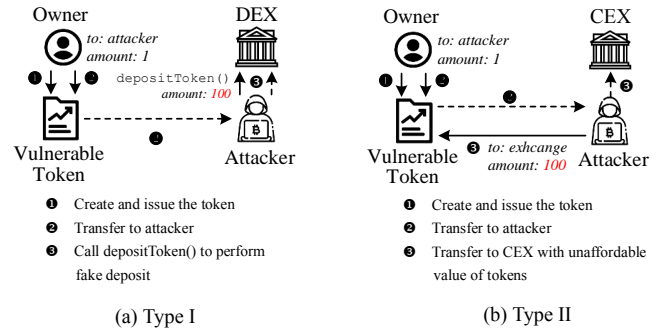


Fig. 5. The fake deposit attack resulted from (a) the incorrect implementation of ERC-20 token and lack of auditing from DEX; (b) the insufficient verification of back-end server of CEX. (Where the solid line is the “Call Flow” and the dash line is the “Money Flow”)

However, in §III-B, we explained two types of flawed verification of DEX and CEX, which requires different deposit behaviors, i.e., different testing process. Therefore, we divided the verifying process into two parts (as shown in Fig. 5), which aim to verify the vulnerabilities existed in DEX and CEX.

Type-I Attack. Type-I attack is related to the DEXes. Under normal circumstances, the `depositToken` in DEXes’ smart contract would invoke the traded token’s `transferFrom` or `transfer`. However, the mis-implementation of functions in ERC-20 tokens would result in Type-I attack. Except for the example raised in §III-B1, there exists some other cases. If the smart contract does not implement the `transferFrom` or `transfer` at all, the invocation from `depositToken` will be automatically executed by the *fallback* function [26], which will be executed if none of other functions matches to the name indicated in the incoming function call. Consequently, some unexpected behaviors would happen.

Therefore, our verifying procedures for Type-I attack are shown in Fig. 5(a). Notice that, we have already deployed the smart contracts of ERC-20 and DEX in the last step. Firstly, we (the owner) issue the corresponding tokens from according to the ERC-20 contract. Then, the owner will transfer only 1 newly issued token to another account, i.e., attacker. Next, the attacker will invoke `depositToken` by calling the DEX to deposit 100 tokens which is definitely unaffordable to attacker. If the token implements the smart contract incorrectly, i.e., not following the ERC-20 guideline like examples depicted in the last paragraph, the deposit request will be handled. Finally, to verify the final result of such a deposit request, we will examine the balance table in the DEX at last. If the attacker has balance which is equal to the amount of tokens deposited into the DEX, the Type-I fake deposit attack is considered successful.

Type-II Attack. The Type-II attack relies on the flaws in back-end source code of CEX that is mentioned in §III-B2. Though it is impossible for us to obtain the source code, we can not underestimate the financial losses which may result from this type of attack. Therefore, we just assume the CEX has a deficiency in validating transactions and perform attacks following the behaviors detailed in Fig. 5(b).

Specifically, like the verification of Type-I attack, we firstly issue the token and transfer only 1 token to attacker account. Then, however, the attacker would call the `transfer` function in the ERC-20 with the `_to` as address of the CEX and `_value` as 100, which is also unaffordable to him. After that, we would examine this transaction in our private chain to check if the `status` is 1. If it is, we can assume this kind of token has possibility to be fake deposited into some defective CEX. Therefore, we have measured the most worst case for the fake deposit behavior between ERC-20 and CEX in Type-II attack.

V. EVALUATION

Our evaluation is driven by the following research questions:

- RQ1 *How many ERC-20 smart contracts are vulnerable to fake deposit attacks, and can be successfully exploited?* We want to measure the overall landscape of fake deposit vulnerability across all the ERC-20 tokens.
- RQ2 *How effective is DEPOSafe in identifying fake deposit vulnerability?* We want to measure the precision and recall of DEPOSafe.
- RQ3 *What is the impact of the fake deposit vulnerability?*

To answer RQ1, we apply the DEPOSafe to all of the 176k ERC-20 smart contracts deployed by the time of this writing. To evaluate the effectiveness of DEPOSafe (RQ2), we manually select samples and perform a fine-grained analysis on the false positives and false negatives introduced by both static detector and dynamic validator in DEPOSafe respectively. To answer RQ3, we further analyze the transactions and volume of these vulnerable ERC-20 tokens and corresponding DEXes, as an indicator to measure the overall impacts.

A. RQ1: Vulnerable ERC-20 Smart Contracts

1) *Overall Result:* After analyzing 176,559 ERC-20 smart contracts deployed before April 2020, 7,735 (4.38%) smart contracts are marked as vulnerable by DEPOSafe. Among them, 56 smart contracts are vulnerable to Type-I attack (see §IV-B3 and Fig. 5), while 7,716 of them can be exploited by the Type-II attack³. It suggests that *fake deposit vulnerability is prevalent in the ERC-20 smart contract ecosystem*.

Note that, for the 56 smart contracts that are vulnerable to Type-I attack, they can be indeed exploited by attackers in the wild, due to their vulnerable implementation. For the 7,716 smart contracts that are vulnerable to Type-II attack, whether the attacks could be successfully performed are also relying on the verification of CEXes. As aforementioned in §III, performing Type-II attack also requires the flawed verification of CEXes, including lacking audit to tradable token and deficiency in back-end verification. As we are unable to acquire the source code of CEXes, we can only measure the vulnerability from the perspective of smart contracts. Nevertheless, these vulnerabilities are indeed existing, which pose great security issues, especially when considering the fake deposit attacks are frequently reported from time to time.

³37 smart contracts can be exploited by both types of attacks.

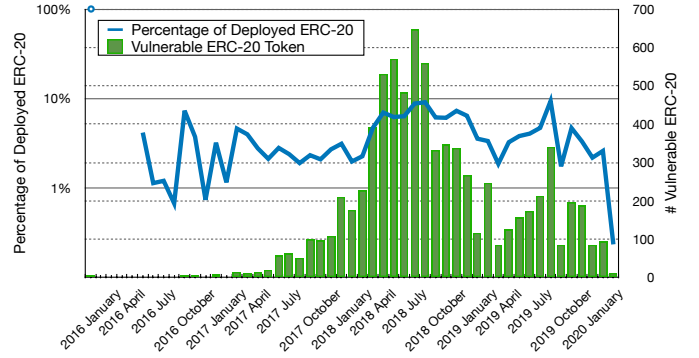


Fig. 6. The distribution of ERC-20 smart contracts with fake deposit vulnerability, and its percentage of deployed ones in each month.

2) *Distribution of the Vulnerable ERC-20 Smart Contracts:* We further analyze the distribution of these 7,735 vulnerable smart contracts based on their deployment time. As depicted in Fig. 6, the number of smart contracts that are vulnerable to fake deposit vulnerability shows an increasing trend till June 2018, which may due to the growing popularity of ERC-20 tokens. After that, the amount of vulnerable contracts declined greatly. We assume that decline is related to a mass of attack behaviors towards Ethereum, which in turn reminds the developers of newly deployed ERC-20 tokens.

Fig. 6 also shows the proportion of the vulnerable ERC-20 contracts within the deployed ones in each month. As we can see, for the first month (January, 2016), all of the 5 ERC-20 tokens are affected, which may due to the developer's unfamiliar to ERC-20 standard. For the following four months, all the deployed ERC-20 contracts are not vulnerable to fake deposit vulnerability. We assume it is because the number of new deployed ERC-20 contracts are small, which are between 15 to 33. From then on, the percentage fluctuated between 1% to 10%, even if the absolute number changes dramatically. This observation suggests although attacks related to this vulnerability have been reported from time to time, there is always a considerable number of developers overlooked it.

B. RQ2: The Effectiveness of DEPOSafe

DEPOSafe is composed of two components: *static detector* and *dynamic validator*. Due to the inherent limitation of static analysis, the false positives will be introduced inevitably. Similarly, though dynamic validator can ensure the exploited smart contracted are real vulnerable, however, it will introduce false negatives, as some vulnerable contracts cannot be easily triggered. Then we seek to manually evaluate the effectiveness of the static detector and dynamic validator, respectively.

Specifically, the first phase of DEPOSafe (i.e., static detector) has reported 9,856 suspicious smart contracts, as they have shown code-level patterns of the vulnerability. As DEPOSafe only reports 7,735 vulnerable contracts after the second phase (i.e., dynamic validator), there remains a gap of 2,121 contracts between these two phases. *Note that, there are no false positives introduced by DEPOSafe, as all the 7,735 reported smart contracts can be exploited successfully.*

1) *Manually Investigation.*: Next, we seek to analyze the 2,121 unreported ones, to see (1) how many of them are false positives introduced by *static detector*, but later removed by our *dynamic validator*; and (2) how many of them are false negatives of DEPOSafe, which are indeed vulnerable ones but missed by the *dynamic validator*. Furthermore, we also sample smart contracts from the unreported ones in the 176K tokens to evaluate (3) the overall coverage of DEPOSafe (as the coverage is relying on the performance of static detector).

For the 2,121 smart contracts, only 671 of them are open-source. Thus, we decide to select contracts from them for achieving accurate manually investigation. We firstly sorted them by the lexicographic order based on the deployed contracts addresses, and then grouped these addresses into 10 subsets (67 contracts in first nine groups, and 68 contracts for the last one). Next, we chose 10 contracts randomly from each subset and analyzed those 100 contracts with source code manually (*Dataset-1*). Moreover, we also randomly select 100 smart contracts (with source code) from the unreported ones in the 176K ERC-20 tokens (*Dataset-2*). In total, we manually investigate 200 smart contracts.

2) *Effectiveness of DEPOSafe*: After two rounds of thorough and careful inspections by the first two authors, we have identified 90 false positive cases introduced by the static detector (but eliminated by dynamic validator) and 10 false negative cases from the 100 sampled smart contracts in **Dataset-1**. As to the **Dataset-2**, all of them are confirmed to be true negatives (i.e., without vulnerability). This result suggests the effectiveness of DEPOSafe, i.e., most of the vulnerable smart contracts can be automatically exploited by DEPOSafe. **Overall, the precision of DEPOSafe is 100% (as all the reported cases are vulnerable), and the recall of DEPOSafe is roughly over 99%⁴.**

3) *False Positives of Static Detector*: The 90 false positives of static detector fall into three categories, including *inter-contract Safemath* (72), *stringent throw check* (15), and *non-standard transfer implementation* (3).

Inter-Contract Safemath. As shown in listing 1, some contracts implement *safeAdd*-like arithmetic operation in *Safemath* contract, which aims to perform an overflow verification by throwing an exception. However, the *Safemath* is often implemented in another contract, and the token contract will inherit it and invoke the functions in it to perform arithmetic operations. For example, the *transfer* in listing 1 calls two functions, i.e., *safeAdd* and *safeSub*, to prevent overflows. Nevertheless, on the bytecode level, the opcode sequence of *safeAdd* (and *safeSub*) can be either inlined into the *transfer* function, or invoked by the opcode *CALL*. In the former case, we can easily locate the *revert* opcode as shown in §IV-A. However, in the latter case, because DEPOSafe does not achieve the inter-contract symbolic execution (due to the limitation of Mythril), we cannot locate

the *revert* operation and conduct the backward tracing accordingly, which will result in false positives.

Stringent throw Check. As we mentioned in §IV-A2 and §IV-A3, in the *transfer* (or *transferFrom*) function, we would firstly locate the variables being updated, and then trace backwards to verify if they are protected by the *throw* statement. However, the semantic information of the bytecode is usually insufficient. As shown in the bottom right node in Fig. 3, we are unable to distinguish the targets of *SSTORE* between *msg.sender* and *_to*. Therefore, to guarantee the soundness of our static detector, we would check the verification on balance table for both of them, which inevitably introduces false positives as only the check for *balanceOf[msg.sender]* is necessary.

Non-Standard transfer Implementation. Even if the *transfer* function is implemented with the correct logic, it may still be implemented in a way that does not conform to the ERC-20 specification. To be specific, as shown in listing 4, the *transfer* invoked *transferFrom* to complete the logic of transferring tokens. Even if such an implementation could check the adequacy of balance of the invoker, this would not be consistent with the specification set out in ERC-20 guidance [3] in which it strictly limit the behaviors of *transfer*, i.e., throw an exception directly inside *transfer* if there is an insufficient balance.

```

1 function transfer(address dst, uint val)
  public returns (bool) {
2   return transferFrom(msg.sender, dst, val);
3 }
4
5 function transferFrom(address src, address dst
  , uint val) public returns (bool)
6 {
7   require(balanceOf[src] >= val);
8   ...
9 }
```

Listing 4. An example of a mis-implementation of *transfer*.

4) *False Negatives of DEPOSafe*: We have observed two reasons leading to the false negatives of DEPOSafe: *insufficient initialization* (9) and *insufficient supply* (1). Notice that, all of them are related to type-II attack (see §III-B2).

Insufficient Initialization. Our dynamic validator follows certain attack pattern that is described in §IV-B to validate the vulnerability. However, some tokens implement non-standard requirement which needs excess initialization. For example, *BigAppleToken* [27], which implements all the mandatory functions in ERC-20 interface, has one special function as shown in listing 5.

```

1 bool public transferEnabled;
2 function enableTransfer(bool _enable) external
  onlyOwner {
3   transferEnabled = _enable;
4   ... // following logic
5 }
```

Listing 5. The snippet in *BigAppleToken*.

⁴It is estimated based on the proportion of false negative cases in Dataset-1 and Dataset-2, as roughly 212 cases (10% * 2121) would be considered to be false negatives, among all the 176K contracts

Apart from that, in the `transfer` function, there is one more line (i.e., `require(transferEnabled)`) before all the statements, which can be regarded as a switch to enable the function.

Moreover, as the function name varies, it is impossible to take every situation into consideration. Therefore, such kind of tokens, which require additional initialization process, cannot pass the validator. Notice that, we will launch a successful fake deposit attack towards it by satisfying this requirement.

Insufficient Supply. Under most circumstances, when the variable `totalSupply` of a contract is initialized, all the balance will be transferred to the owner’s account, i.e., `msg.sender`. However, in some cases, the initial supply is given to a designated address, like the code snippet shown in listing 6. As it is impossible to assign the addresses of new created accounts even in the testnet of Ethereum, we cannot execute the following logic of dynamic validator unless modifying the source code.

```
1 address public founder = 0xCB7E...; // a fixed
   address
2 balances[founder] = totalSupply;
```

Listing 6. An example of the insufficient supply problem.

Moreover, in some cases, there might not exist initial supply at all. Thus we cannot perform the validation shown in Fig. 5(b).

C. RQ3: The Impact of the Fake Deposit Vulnerability

We further measured the impact of the influenced tokens. For those 56 tokens that can be com

As for those 7,716 that are under Type-II attack pattern, they are potentially at risk of being attacked in the event of insufficient validation of CEXes. We therefore assessed the top 10 tokens with the highest market capitalization affected by Type-II attack. As shown in Table I, these top 10 ERC-20 tokens account for over 984 million USD cap. Note that, here we consider fully diluted market cap, which is calculated by multiplying the token total supply with the current market price per token. The data is acquired on 10th June from etherscan.io. Moreover, they have over 591K holders and 4 million pieces of transaction records in total. Therefore, if a CEX allows these tokens to be traded on its platform without comprehensive verification, the financial loss will be tremendous.

VI. DISCUSSION

Mitigation of Fake Deposit Vulnerability For developers, strictly following the official guideline [3] is always a good choice. To be specific, developers should implement all the six mandatory functions carefully, especially the `transfer` and `transferFrom` that are closely related to the transferring behaviors. The developer should *throw* an exception if any mis-behavior happens (like insufficient balance or allowance) in both of them instead of returning a boolean value. For the DEX, as both of the `transfer` and `transferFrom` will return a boolean value to indicate if the transferring succeeds, the DEX should never assume the return value

TABLE I
TOP 10 VULNERABLE TOKEN (TYPE-II) WITH MOST CAP.

Token	Cap (\$)	# Holders	# Transactions
BAT	387,569,543	304,637	2,054,172
BRC	391,575,806	87,118	1,382,950
POWR	27,959,127	49,659	377,958
HPT	62,665,797	1,237	7,317
NULS	13,898,085	3,748	44,962
RPL	39,988,851	3,006	30,266
AOA	25,404,742	101,078	146,523
WIX	20,910,634	270	2190
MRPH	10,258,018	4,099	38,308
BCV	4,347,202	36,696	203,087
Total	984,577,805	591,548	4,293,733

from tokens. Additionally, it should handle the exception properly if the token raised it. For the CEX, it should perform a comprehensive verification on each transaction of deposit request in back-end servers.

Limitation We characterize the fake deposit vulnerability mainly from the perspective of the ERC-20 smart contracts. As we mentioned, performing the Type-II attack also requires the flawed verification of CEXes. However, we are unable to get the source code of CEXes. Thus, we only measure the scale of the vulnerable ERC-20 tokens, which is an upper bound for the Type-II attack. Besides, we did not analyze the transactions on Ethereum, which may reveal the existence of real-world fake deposit attacks. Furthermore, it was reported that the fake deposit attacks have been observed in other blockchains (e.g., USDT [28], EOSIO [29]), while this paper only focused on Ethereum. We leave them for the future work.

VII. RELATED WORK

Characterizing Blockchain Ecosystem. Due to the popularity of the cryptocurrency, several works have been published to measure the blockchain ecosystem [30]–[35]. For example, [31] presented a systematic study to characterize the code reuse practice in the Ethereum ecosystem. [35] characterized money transfer, contract creation and invocation through graph analysis. Apart from Ethereum, [30] mainly focused on the activities in EOSIO blockchain, and [34] proposed an approach to identify entities in Bitcoin blockchain.

Program Analysis of Smart Contract. Based on several program analysis method, e.g., symbolic execution, formal verification, lots of frameworks have been proposed to improve the security of smart contracts [36]–[43]. Specifically, M. Mossberg et al. [41] proposed Manticore, a dynamic symbolic execution framework for analyzing Ethereum smart contract. Madmax [36], implemented by N. Grech et al., has mainly concentrated on gas-related vulnerabilities. Additionally, [43] implemented a symbolic execution framework for analyzing smart contracts written in WebAssembly, which is used by EOSIO blockchain and will be further adopted by Ethereum.

ERC-20. Several works [44]–[50] have focused on the ERC-20-related behaviors and smart contracts in Ethereum. Specifically, [45], [46], [49] analyzed the network of transfer tokens

between accounts in Ethereum. Except for the traditional graph analysis on Ethereum, S. Somin et al. [48] presents the analysis of the dynamical properties of the ERC20 protocol, which in turn allows the prediction of some network parameters. Additionally, T. Chen et al. proposed TokenScope [50], which is able to detect transactions that triggers inconsistent behaviors.

VIII. CONCLUSION

In this work, we have systematically characterized the fake deposit vulnerability in Ethereum. DEPOSafe, an automated tool is proposed to perform the detection and verification of the vulnerability. Experiment results suggest the efficiency of DEPOSafe. Our observations indicate the prevalence of fake deposit vulnerability in the ERC-20 smart contracts. Our efforts can positively contribute to bring developer awareness, attract the focus of the research community and regulators, and promote best operational practices across blockchains.

REFERENCES

- [1] (2020, Apr.) Bitcoin official site. [Online]. Available: <https://bitcoin.org/>
- [2] (2020, Apr.) Ethereum official site. [Online]. Available: <https://ethereum.org/>
- [3] (2019, Mar.) Official guide to ERC-20 standard. [Online]. Available: <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md>
- [4] Julia Magas. (2019, June) Explanation of DEX. [Online]. Available: <https://cointelegraph.com/explained/dex-explained>
- [5] Comply Advantage. (2020, Apr.) Definition of DEX. [Online]. Available: <https://complyadvantage.com/knowledgebase/kyc/>
- [6] Mix. (2018, Nov.) Attacks towards Ethereum exchange. [Online]. Available: <https://thenextweb.com/hardfork/2018/11/22/vulnerability-ethereum-gastoken-exchange/>
- [7] Adrian Zmudzinski. (2019, Sep.) Attacks towards Ethereum exchange. [Online]. Available: <https://cointelegraph.com/news/developers-of-ethereum-dex-protocol-airswap-disclose-critical-exploit>
- [8] —. (2019, July) Attacks towards Ethereum exchange. [Online]. Available: <https://cointelegraph.com/news/0x-dex-protocol-suspended-because-of-vulnerability-funds-safe>
- [9] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum,"
- [10] CHINABTCNEWS. (2018, July) Attacks against fake deposit vulnerability. [Online]. Available: <https://news.8btc.com/slowmist-3619-eth-based-tokens-are-affected-by-fake-deposit-vulnerability>
- [11] SyncFab. (2018, Aug.) Attacks against fake deposit vulnerability. [Online]. Available: <https://medium.com/syncfabmf/erc20-fake-deposit-vulnerability-countered-by-bitforex-maintenance-and-mfg-smart-contract-upgrade-6699bf086e28>
- [12] Stephen O'Neal. (2019, Mar.) Attacks against fake deposit vulnerability. [Online]. Available: <https://cointelegraph.com/news/ledger-client-address-issue-and-fake-deposits-community-spots-two-vulnerabilities-related-to-monero>
- [13] (2020, May) Attacks against fake deposit vulnerability. [Online]. Available: <https://www.chainnews.com/news/524364903562.htm>
- [14] (2020, Apr.) Official site of mythrill. [Online]. Available: <https://github.com/ConsensSys/mythrill>
- [15] J. R. Cruz. (2013, May) Keccak256 hash function. [Online]. Available: <https://www.drdoobs.com/security/keccak-the-new-sha-3-encryption-standard/240154037>
- [16] (2020, Apr.) Documentation of Solidity. [Online]. Available: <https://solidity.readthedocs.io/en/v0.6.6/>
- [17] ethersvm. (2019, Sep.) Opcodes supported by EVM. [Online]. Available: <https://ethersvm.io/>
- [18] (2020, Apr.) Coinbase official site. [Online]. Available: <https://www.coinbase.com/>
- [19] (2020, Apr.) Binance official site. [Online]. Available: <https://www.binance.com/en>
- [20] (2020, Apr.) EtherDelta official site. [Online]. Available: <https://etherdelta.com/>
- [21] (2020, Apr.) IDEX official site. [Online]. Available: <https://idex.market/>
- [22] (2018, July) Fake deposit attack. [Online]. Available: <https://www.chai-news.com/articles/168840801152.htm>
- [23] (2020, June) web3.js, the Ethereum JavaScript API. [Online]. Available: <https://github.com/ethereum/web3.js/>
- [24] (2020, June) Ganache CLI provides private chain for Ethereum. [Online]. Available: <https://github.com/trufflesuite/ganache-cli>
- [25] (2020, Apr.) Data explorer for Ethereum. [Online]. Available: <https://etherscan.io/>
- [26] D. Crescenzi. (2018, June) Definition of fallback function. [Online]. Available: <https://medium.com/upstate-interactive/the-truth-about-fallback-functions-in-solidity-a2c604f8e66b>
- [27] (2020, June) BigAppleToken. [Online]. Available: <https://etherscan.io/address/0xDfbfCe00fc2fDBd5951f0a1bc4D49b6d87BF516D#code>
- [28] (2020, Mar.) Fake deposit attack towards USDT. [Online]. Available: <https://www.coinness.com/news/582082>
- [29] SlowMist. (2019, Mar.) Fake deposit attack towards EOSIO. [Online]. Available: <https://medium.com/@slowmist/hard-fail-status-attack-for-eos-7cfa73ae7d4b>
- [30] Y. Huang, H. Wang, L. Wu, G. Tyson, X. Luo, R. Zhang, X. Liu, G. Huang, and X. Jiang, "Characterizing eosio blockchain," *arXiv preprint arXiv:2002.05369*, 2020.
- [31] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," *arXiv preprint arXiv:1905.00272*, 2019.
- [32] M. Saad, J. Choi, D. Nyang, J. Kim, and A. Mohaisen, "Toward characterizing blockchain-based cryptocurrencies for highly accurate predictions," *IEEE Systems Journal*, 2019.
- [33] P. Xia, B. Zhang, R. Ji, B. Gao, L. Wu, X. Luo, H. Wang, and G. Xu, "Characterizing cryptocurrency exchange scams," *arXiv preprint arXiv:2003.07314*, 2020.
- [34] M. Jourdan, S. Blandin, L. Wynter, and P. Deshpande, "Characterizing entities in the bitcoin blockchain," in *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2018, pp. 55–62.
- [35] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1484–1492.
- [36] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.
- [37] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *NDSS*, 2018, pp. 1–12.
- [38] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.
- [39] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.
- [40] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.
- [41] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.
- [42] P. Tsankov, A. Dan, D. Drachler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.
- [43] N. He, R. Zhang, L. Wu, H. Wang, X. Luo, Y. Guo, T. Yu, and X. Jiang, "Security analysis of eosio smart contracts," *arXiv preprint arXiv:2003.06568*, 2020.
- [44] G. Fenu, L. Marchesi, M. Marchesi, and R. Tonelli, "The ico phenomenon and its relationships with ethereum smart contract environment," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 26–32.
- [45] F. Victor and B. K. Lüders, "Measuring ethereum-based erc20 token networks," in *International Conference in Financial Cryptography and Data Security*. Springer, 2019, pp. 113–129.

- [46] S. Somin, G. Gordon, and Y. Altshuler, "Network analysis of erc20 tokens trading on ethereum blockchain," in *International Conference on Complex Systems*. Springer, 2018, pp. 439–450.
- [47] M. Fröwis, A. Fuchs, and R. Böhme, "Detecting token systems on ethereum," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 93–112.
- [48] S. Somin, G. Gordon, A. Pentland, E. Shmueli, and Y. Altshuler, "Erc20 transactions over ethereum blockchain: Network analysis and predictions," *arXiv preprint arXiv:2004.08201*, 2020.
- [49] W. Chen, T. Zhang, Z. Chen, Z. Zheng, and Y. Lu, "Traveling the token world: A graph analysis of ethereum erc20 token ecosystem," in *Proceedings of The Web Conference 2020*, 2020, pp. 1411–1421.
- [50] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1503–1520.