SMARTGIFT: Learning to Generate Practical Inputs for Testing Smart Contracts

Teng Zhou*, Kui Liu*†‡, Li Li§, Zhe Liu*, Jacques Klein¶ Tegawendé F. Bissyandé¶
*Nanjing University of Aeronautics and Astronautics, Nanjing, China
{tengzhou, kui.liu, zhe.liu}@nuaa.edu.cn
† Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Nanjing, China
‡State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
§Monash University, Melbourne, Australia, li.li@monash.edu
¶Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
{jacques.klein, tegawende.bissyande}@uni.lu

***Abstract*—With the boom of Initial Coin Offerings (ICO) in the financial markets, smart contracts have gained rapid popularity among consumers. Smart contract vulnerabilities however made them a prime target to malicious attacks that are leading to huge losses. The research community is thus applying various software engineering technologies to smart contracts to address them. In general, to detect vulnerabilities in smart contracts, mutation and fuzz based testing approaches have been widely studied and indeed achieved promising performance on benchmark datasets. Generating test inputs with mutation approaches essentially relies on the available test cases in a smart contract program. In our preliminary study, however, we observed that 56.4% of 218 identified open-source smart contract project repositories do not provide any test case for validation. Fuzzing test inputs leads to random values and lacks practical usefulness. Our work addresses this problem: we propose an approach, SMARTGIFT, which generates practical inputs for testing smart contracts by learning from the transaction records of real-world smart contracts. Leveraging a collected set of over 60 thousand transaction records, SMARTGIFT is able to generate relevant test inputs for ∼77% smart contract functions, largely outperforming the traditional fuzzing approach (successful for only 60% functions). We further demonstrate the practicality of the test inputs by using them to replace the test inputs of the ContractFuzzer state of the art smart contract vulnerability detector: with inputs by SMARTGIFT, ContractFuzzer can now detect 131 of the 154 vulnerabilities in its benchmark.**

***Index Terms*—Test Input Generation, Smart Contract, Deep Learning.**

I. INTRODUCTION

Blockchain [1] technology has attracted a lot of attention when it achieved a significant milestone with the implementation of decentralized and distributed digital ledgers for recording transactions of cryptocurrencies [2] (e.g., Bitcoin [3], Ethereum [4] and Litecoin [5]). In this vein, *smart contracts*, which leverage blockchain technology to implement a computerized transaction protocol that executes the terms of a contract, are increasingly investigated by the industry and research communities [6], [7]. Smart contracts are provided for participants to reduce the need for trusted intermediates when initiating contracts so as to reduce possible fraud losses, arbitration and enforcement costs, as well as malicious or accidental exceptions [8].

As a self-executing program without the need for any external trusted authority, a smart contract is promising for various interactions that are related to data-driven transactions [1], [7]. At the end of 2018, 5-10% of jobs advertised on the popular Guru[1] platform for freelancers were related to smart contracts and blockchain [9]. While many countries investigate the actual legal value of smart contracts in court, more and more developers are devoting their careers to boosting the application of smart contracts in various domains, such as finance, gaming, and notary [10]. In Ethereum[2], a global, open-source platform for decentralized applications, over two million smart contract accounts[3] were deployed at the end of September 2020, counting for a total of more than 110 million Ether that is worthy of over 37.4 billion US dollars (Ether is the digital money and the currency of Ethereum apps, and one Ether was worth more than 340 US dollars at that time).

Security problems in smart contracts can result in tremendous financial losses in the real world. For example, the notorious "DAO" attack on the reentrancy problem caused DAO's investors to lose 3.5 million Ether (at least 60 million US dollars at that time) in June 2016 [11]. Similarly, the team behind the Parity Ethereum software client revealed how a critical code flaw (e.g., the Parity Freeze) resulted in the freezing of $160 million worth of Ether [12]. These costly attacks demonstrate that it is essential to improve the quality of smart contracts before deploying them, especially since they are immutable once deployed. Unfortunately, it is an extremely difficult task to develop trustworthy and safe smart contracts because of the complex semantics of the underlying domain-specific languages (e.g., Solidity[4]) as well as the potentially-breaking changes in their evolution. To address the security issues that smart contracts can carry in their programs, various approaches have been proposed in the literature [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25],

*Kui Liu and Zhe Liu are the corresponding authors.

[1]https://www.guru.com
[2]https://ethereum.org
[3]https://etherscan.io/accounts
[4]https://solidity.readthedocs.io

DOI 10.1109/ICSME52107.2021.00009

[26]. A large number of these works build approaches that are about generating test inputs for further testing smart contracts, in order to identify potential vulnerabilities before deployment.

In the literature, the generation of test inputs for smart contracts mainly relies on fuzzing and mutation [27], [28], [25], [29], [30], [31], [32], [33]. For example, Jiang et al. [34] proposed to build seed inputs with the valid input domain and the inputs frequently used by some data types in smart contracts, to further fuzz inputs for testing ABI of smart contracts. To detect reentrancy vulnerabilities in smart contracts, ReGuard [35] performs fuzz testing for smart contracts by iteratively generating random but diverse transactions. More recently, Ma et al. [36] explored the feedback-directed mutation and the parameter types of smart contract functions to fuzz test inputs. By analyzing Solidity documents and issues about smart contracts listed in GitHub and Stack Exchange, Liu et al. [37] defined 15 mutation operators for test input generation. Andersta et al. [38] designed ten classes of mutation operators inspired by real faults in Solidity smart contracts to generate new test inputs.

The state-of-the-art test case (i.e., input) generation for smart contracts, however, ❶ is highly dependent on the quality of existing inputs, and ❷ involves a high level of randomness that could impact the effectiveness of generated test inputs. In addition, ❸ the test inputs provided in smart contract test suites are, most of the time, impractical since they are set with simple values or are randomly provided by developers. For example, Figure 1 shows an example of a test for a smart contract function excerpted from a Decentralized Application (DApp) named SmartContractSlackDapp[5] implementing a lottery via Slack. In this example, the function *placeBets()* is tested by checking whether the pot will be increased successfully after two players place a bet, where both bets are set with "`minStake`". The value of `minStake` in the test case is unfortunately rather unrealistic because of the arbitrary or random setting. Such a test case does not cover the smaller (and realistic) bet that would test the minimum limit or the maximum value that would cause an overflow exception. Finally, ❹ in our preliminary study, we find that the majority of smart contract open source projects do not include any test cases to bootstrap further input generation.

Overall, there is still room for improvement in the current testing practice of smart contracts. In particular, developers genuinely need practical inputs for testing their smart contracts. Our intuition is that we can learn from the real-world transactions that are made on deployed platforms to better tailor the input space for test case generation. Fortunately, numerous smart contracts have been deployed on the decentralized blockchain platforms (e.g., Ethereum), of which various actual transactions can provide abundant practical and actionable inputs for executing smart contracts. Recent work by Liu et al. [39] on identifying method naming bugs has shown that similar tasks are independently implemented and named similarly by different developers. Smart contract

[5]https://github.com/senacor/SmartContractSlackDapp September, 2020

```
// Function placeBets excerpted from SmartContractSlackDapp.
function placeBets() public payable sufficientFunds
     gameOngoing {
  pot += msg.value;
  participants.push(msg.sender);
  UserPutBets(msg.sender, pot);
}

// Testing code
var minStake = 1000000000000;
...
describe('placeBets', function() {
  it("should increase pot upon placing a bet", function() {
    return lot.placeBets({from:accounts[1], value: minStake
     }).then(function() {
      return lot.getPot.call();
    }).then(function(actual) {
      currPot += minStake;
      assert.equal(actual, currPot, "bet could not be placed
      correctly")});});

  it("should increase pot again upon placing another bet",
     function() {
    return lot.placeBets({from:accounts[2], value: minStake
     }).then(function() {
     return lot.getPot.call();
    }).then(function(actual) {
      currPot += minStake;
      assert.equal(actual, currPot, "bet could not be placed
      correctly")});});});
```

Fig. 1. Example of a test case excerpted from a lottery DApp.

implementations should therefore share some similarities in functionalities. Therefore, **we build on the hypothesis that the similar functions of smart contracts can be validated with similar inputs**. In this paper, we propose generating practical test inputs for newly-written smart contracts by learning from the successfully-deployed and executed test cases of smart contracts in the real world. **We define a practical input as a meaningful input for testing: it is a realistic one used by users in practice, and is likely to offer better coverage for discovering vulnerabilities.**

Our work investigates the aforementioned hypothesis and builds a learning approach to improve smart contract testing with better inputs. To that end, we first build a dataset of smart contract functions with practical inputs by collecting the transaction records of smart contracts successfully deployed in the real world. Each function is processed to obtain its signature that is further used to yield an embedding (i.e., a numerical vector) with the BERT [40] deep representation learning technique. Given a new smart contract, all of its functions will be processed in the same way to extract their features, which will be used to match similar functions from the collected dataset. The practical inputs of the matched similar functions are considered to derive the test inputs for the functions of the smart contract under test. Finally, we implement a prototype tool named SMARTGIFT (**G**enerating practical **I**nputs **F**or **T**esting **Smart** contracts) to generate practical inputs for testing smart contracts. We also perform a detailed evaluation about the effectiveness of our approach. The experimental data is publicly available at: **https://github.com/chaoweilanmaohahaha/SmartGift**

This paper makes the following contributions:

1) We conduct an investigation on the availability of test cases in open-source smart contract projects. Our main

finding is that test cases are rare, which is a concern about their validation before (immutable) deployment.

2) We present an approach to generate practical inputs for testing smart contracts by learning from concrete inputs of functions. These inputs are extracted from the transaction records of executed real-world smart contracts.
3) We implement a prototype tool, SmartGift, which is evaluated through three experiments that build on 66,528 transaction records, 145 smart contract functions, and 154 vulnerable smart contracts. ❶ The experimental results show that the test inputs derived from other smart contracts by SmartGift are realistic to test $\sim$77% functions in the smart contracts under test. ❷ Test inputs from SmartGift are capable of uncovering issues in smart contracts by covering relevant corner cases in the programs. ❸ Finally, SmartGift was able to generate inputs that led to the detection of 131 (out of 154) benchmark vulnerabilities with ContractFuzzer [34].

II. Background and Motivation

Before introducing how to generate test inputs, this section presents some basic knowledge of smart contracts and a preliminary study that motivates this work.

A. Smart Contract

The "Smart Contract" concept was popularized by Nick Szabo [8]. In the industry, a smart contract is a self-executing contract with the terms of the agreement between customer and seller being directly written into pieces of code, without the need for a central authority, legal system, or external enforcement mechanism. The code and the agreements contained therein exist across a distributed, decentralized blockchain network. The code controls the execution, and transactions are trackable. Different from the scripts used in Bitcoin, the programming language of smart contracts is Turing-completed, which means users can achieve more complex goals and logics such as crowdfunding, gaming, etc.

The procedure of developing smart contracts is different from the development of programs with commonly used programming languages (e.g., C, C++, Java, Python). After writing and compiling a smart contract to the bytecode, it will be deployed to the distributed, decentralized blockchain network. Once a smart contract is deployed on the blockchain, it cannot be modified anymore even though it has a fatal vulnerability that could be exploited by attacking incidents until the contract is destructed. The interaction between smart contracts is proceeded with their transactions that carry the function name and parameters in its data area. Solidity is the most popular programming language for developing smart contracts, its grammar is similar to Javascript.

Solidity was firstly proposed in 2014, and was developed by the Ethereum project's Solidity team for developing smart contracts. As a new proposed programming language, Solidity has been evolving with 78 versions from v0.1.2 to v0.7.3 in recent 5 years. The fast update rate of Solidity could be one of the reasons that its smart contracts could be unreliable. It thus needs to test smart contracts with effective and practical test inputs to detect the potential defects before deploying.

TABLE I
Smart contract repositories w/o test case.

# Repositories	# Repositories without Test Case
218	123

TABLE II
A survey about 11 contract projects.

Contract Name	# Test Files	LoC of Test Code
FBT	1	377
CT	3	233
MT	5	758
ENS	6	318
GMSW	6	597
ERCF	6	1,792
RNTS	6	1,079
VT	7	939
BNK	13	7,405
MR	15	3,190
HPN	53	32,819

B. Preliminary Study

As a preliminary study on investigating to what extent the test cases are prevalent in smart contracts, we resort to the open-source smart contract projects hosted on GitHub.

1) Test Cases in Real-World Smart Contracts: We first assess how many real-world smart contracts are provided with/without test suites for their validation. We collect smart contracts from GitHub by searching with the keyword "*smart contract*"[6]. We first select the top-500 *Best Match* repositories. After checking these repositories manually, 218 repositories are indeed related to smart contracts written with Solidity[7]. Overall, as presented in Table I, we observe that $\sim$56.4% (=123/218) smart contracts are not attached to any test case. It indicates that, on the one hand, smart contracts should be tested for their robustness. It is in line with the traditional normal programs (e.g., C or Java programs), although they are the new-emerged decentralized applications. On the other hand, a lot of smart contract projects are not validated with any test suite, which means they cannot find the potential vulnerable issues that could make smart contracts unreliable.

We further closely look at the test cases provided in smart contract projects. We select a benchmark dataset, built by Wustholz et al. [27] for testing smart contracts, which includes 17 projects that are popular projects carefully picked from the Ethereum community and Github. Six of them do not contain any test cases, the remaining 11 projects shown in Table II are considered in this investigation. The data presented in Table II, and in particular the relatively high number of lines of code to write tests, suggest that the testing of smart contracts is indeed a non-trivial activity that requires developer manpower. After checking the code of all the test cases in the 11 projects manually, we conclude several observations presented below:

[6]The searching was conducted on November 30, 2020.
[7]Solidity is the most used language to specify smart contracts. We thus restrict our search to Solidity.
[8]https://github.com/ConsenSys/Tokens

```
it (``milestones tests``, function (){
   return ModumToken.deployed().then(function (instance) {
   }).then(function (retVal) {
      return utils.testMint(contract, accounts, 0, 1001,
         1000)
   }).then(function (retVal) {
      return utils.testVote(contract, accounts, 900000,
         1001, 1000, 0, true, false, 900000);
   }).then(function (retVal) {
      return utils.testVote(contract, accounts, 3000000,
         1001, 1000, 0, true, false, 3900000);
   }).then(function (retVal) {
      return utils.testVote(contract, accounts, 3000000,
         1001, 1000, 0, true, false, 6900000);
   }).then(function (retVal) {
      return utils.testVote(contract, accounts, 3000000,
         1001, 1000, 0, true, false, 9900000);
   }).then(function (retVal) {
      return utils.testTokens(contract, accounts, 0, 9900000
         + 2001, 9900000, 1001);
   }).catch((err) => {throw new Error(err) });
});
```

Fig. 2. Manually Specified Seeds in the Smart Contract[8].

```
it(`creation: should succeed in creating over 2^256 - 1 (max
     ) tokens`, async () => {
   // 2^256 - 1
   const HST2 = await EIP20Abstraction.new(
      `115792089237316195423570985008687907853269984665640564
      0564039457584007913129639935`,
      `Simon Bucks`, 1, `SBX`, { from: accounts[0] });
   const totalSupply = await HST2.totalSupply();
   const match = totalSupply.equals(
      `1.15792089237316195423570985008687907853269984665640564
      0564039457584007913129639935e+77`);
   assert(match, `result is not correct`);
});
```

Fig. 3. Boundary Case Checked in the Smart Contract[9].

(1) All of these 11 projects have tests with manually-specified inputs. Manual specification (exemplified in Figure 2) requires that developers have a comprehensive understanding and consideration for all potential cases, which will spend high costs. The specification could be insufficient (e.g., some practical cases are not considered) that would limit the scale of testing and increase the testing costs.

(2) Checking the boundary conditions of the code logic is considered by most smart contract developers. For example, some test cases are designed to detect the numerical overflow (shown in Figure 3). However, some developers failed to consider such validation for their contracts.

(3) Developers prefer to test "normal" scenarios with various test cases. Only few developers consider corner cases.

2) Similarity of Real-World Execution Inputs: Execution inputs represent the parameter values of Smart contract transactions executed in real-world use case scenarios. Since our hypothesis is that we can leverage inputs from one smart contract to drive the generation of inputs of others (which have fewer test cases), we propose to investigate the actual similarity of execution inputs of smart contracts in the wild. To that end, we consider the transaction records from Etherscan[10], the Ethereum Blockchain Explorer. Collection details are provided in Section IV-B. We are indeed able to confirm that a large proportion ($\sim$60% cases, cf. Section V-A) of inputs (excluding addresses) are the same across smart contracts.

[9]https://github.com/ScJa/ercfund
[10]https://etherscan.io/

To sum up, while test cases for validating smart contracts are important in development, many development teams fail to ensure test suites are available for their smart contract programs. Additionally, the manual efforts for test specifications generally yield inputs with limited potential to inefficiently and ineffectively uncover potential issues in smart contracts. We hypothesize that the generation of test inputs for new-developed smart contracts could be benefitted from past transactions in the real world. To mitigate the challenge in building test suites for smart contracts, we propose to automatically generate practical (i.e., meaningful) test inputs for smart contracts by learning from real-world cases.

III. SMARTGIFT - GENERATING INPUTS FOR TESTING SMART CONTRACTS

In this paper, we propose an approach to automatically generating practical test inputs for smart contracts by leveraging a deep representation learning technique to learn the inputs of similar executing functions from the transaction records of real-world smart contracts.

SMARTGIFT consists of three basic steps: data preprocessing, representation embedding, and test input generation, as presented in Figure 4. The functions in smart contracts are processed by extracting their signatures that are further tokenized to feed a representation embedding model (i.e., in this case BERT [40]). With the embedded representation, SMARTGIFT is capable of selecting the similar functions by calculating the similarities between the embedded signatures, that will finally be used to generate the test inputs for the functions of given smart contracts.

A. Data Pre-processing

This phase aims at preparing the raw data of the given functions in smart contracts to be fed into the workflow of SMARTGIFT. A function in a smart contract is represented with a function name, parameter declaration(s) and executable body code. Previous works (e.g., [39]) have shown that the combination of the function name and the parameters can be a good approximation of the summarized semantic description of each function. In this paper, we consider the combination of the function name and the parameter types as the signature of each function. For example, the function shown in Figure 5 could be represented with a function signature "`updateTokenPrice, uint`", where the function name is "updateTokenPrice" and the parameter type is "uint".

Developers have different habits on the naming convention of functions, which could impact the representation learning of function signatures. To resolve this problem, we tokenize the function signature with the camel case and underscore naming conventions, that are widely used to split code identifiers in the literature. Additionally, all tokens of signatures are converted into lowercase letters. Concretely, when we meet the camel-case like "`updateTokenPrice`" shown in Figure 5, we split it into a token sequence: `[update, token, price]`. Finally, a function signature is thus represented as

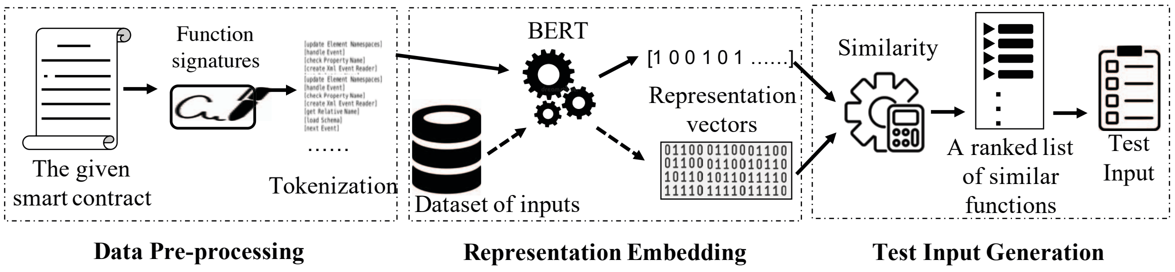

Fig. 4. The overview of SMARTGIFT.

```
function updateTokenPrice(uint _newTokenPrice) public
    onlyOwner {
  tokenPrice = _newTokenPrice;
  return;
}
```

Fig. 5. An Example Function Description in Smart Contract.

a sequence of signature tokens which are suitable for the deep representation learning described in Section III-B. For example, we process the function signature in Figure 5 to be `[update, token, price, uint]`.

B. Representation Embedding

This step aims at producing a numerical representation of a function signature. This numerical representation is further leveraged to compute similarities between functions. We remind that to yield practical inputs for a given function of a smart contract, the overall idea is to ”copy” the inputs of real-world similar functions. When the deep representation learning is applied to the training data, it produces embedding models that have learned to embed all tokens of the training data into numerical representation vectors. These vectors are also referred to as embeddings [41].

In the scenario of this paper, we consider an embedding model, BERT [40], that initially targets natural language data in terms of the learning algorithm and training data. BERT requires large datasets for training the embedding model. As it is now custom in the literature, we instead leverage a pre-trained 24-layer BERT model, which was trained on a Wikipedia corpus, to embed the tokenized function signatures. Token sequences of function signatures are embedded into vectors with BERT since token sequences of function signatures resemble descriptive sentences.

C. Test Input Generation

With the embedded representations of function signatures, SMARTGIFT can match the similar functions for the given smart contract by calculating the cosine similarities of their numeric vectors. The practical inputs of the similar functions will be considered as the test inputs for the function of the smart contract under test.

In the preliminary study, we observe that some functions can have similar identifiers (i.e., same function name), but different parameter data type names for the same concept. This inconsistency could impact the effectiveness of matching adequate inputs for a contract under test. To resolve such a problem, we propose to classify the results sorted by similarity into three categories according to the data types of parameters, which is detailed in Algorithm 1:

Algorithm 1: Classifying the sorted functions into three categories.

```
Input  : $F_s$, a set of similar functions sorted by similarities.
Input  : $f$, the given function.
Output : $F_1$, a set of similar functions in the first category.
Output : $F_2$, a set of similar functions in the second category.
Output : $F_3$, a set of similar functions in the third category.
1  Function classify ($F_s$,$f$)
2      foreach $f_s \in F_s$ do
3          if sameDataTypes ($f_s$.argTypes, $f$.argTypes) then
4              $F_1$.add($f_s$);
5          else if compatibleDataTypes ($f_s$.argTypes, $f$.argTypes) then
6              $F_2$.add($f_s$);
7          else
8              $F_3$.add($f_s$);
9  Function sameDataTypes ($f_s$.argTypes, $f$.argTypes)
10     foreach argType $\in$ $f$.argTypes do
11         if $f_s$.argTypes.contains(argType) then
12             $f_s$.argTypes.remove(argType);
13         else
14             return false;
15     return true;
16 Function compatibleDataTypes ($f_s$.argTypes, $f$.argTypes))
17     foreach $argType_1 \in f$.argTypes do
18         if $f_s$.argTypes.contains($argType_1$) then
19             $f_s$.argTypes.remove($argType_1$);
20         else
21             isCompatible ← false;
22             foreach $argType_2 \in f_s$.argTypes do
23                 if isCompatible ($argType_1$, $artType_2$) then
24                     $f_s$.argTypes.remove($argType_2$);
25                     isCompatible ← true;
26                     break;
27             if !isCompatible then
28                 return false;
29     return true;
```

- **C1**: The first category strictly constrains that both the parameters of the given function and the parameters of its similar functions must have the same data types. As presented lines 9-15 in Algorithm 1, when the data types of all parameters for the given function are contained in the data types of parameters of the similar function, such a similar function will be classified into this category. Function name and parameter data types in the function signatures are named by developers based on their coding style, knowledge and understanding [39]. It is inevitable for developers to use different data type names for the same concept, which could noise the representation learn-

```
[
 {
  func: {
   function_name: transfer,
   parameters: [{name: to, type: address},{name: value, type
    : uint256}]
  }
  C1-func_inputs:[{
   similar_func: {
    function_name: transferTo,
    parameters: [{name: to, type: address}, {name: value,
     type: uint256}]
   },
   inputs:[{0x3d57f6f449a35ae49ad11cc2643a047b73a548b4,
     3244999999999999737856}, ...]
  }]
 }
]
```

Fig. 6. Example of generated C1 inputs for testing a function.

```
[
 {
  func: {
   function_name: setExchangePrice,
   parameters: [{name: price, type: int256}]
  }
  C1-func_inputs:[],
  C2-func_inputs:[{
   similar_func: {
    function_name: setTokenPrice,
    parameters: [{name: _tokenPrice, type: uint256}]
   },
   inputs:[{3233454417711841}, ...]
  }]
 }
]
```

Fig. 7. Example of generated C2 inputs for testing a function.

ing and similarity calculation. Therefore, this category aims at purifying the similar functions suggested by similarities. Figure 6 presents an example of generated inputs in C1 category for testing the function `transfer`.
- **C2**: This category relaxes restrictions of parameter data types. In our preliminary study, we observe that there are upward-compatible data types in Solidity smart contracts, such as from `uint128` to `uint256`. In some cases, the values for `uint256`-type parameters can be downward-compatible to the `uint128`-type parameters when the values do not overflow the `uint128`. And in other cases data can be used between `uint` and `int` at the same time. However, the first category cannot cover such specific situations. To resolve this limitation, we propose this category with relaxed restrictions on data types. Figure 7 shows an example of generating C2 test inputs.
- **C3**: The third category considers the remaining similar functions that have not been classified in C1 or C2. For some specific functions, the previous two categories (i.e., C1 and C2) cannot seek out any similar functions for them. Thus, SMARTGIFT relies on the similarities to recommend functions for them. Figure 8 presents an example of generated inputs in C3 category for testing the function `move`.

In practice, to generate the test inputs of a given smart contract under test, we consider the practical inputs (i.e., execution inputs) of the functions present in Category C1. If the number of inputs generated from functions in C1 is not sufficient, we consider C2 and finally C3. In this study, we

```
[
 {
  func: {
   function_name: move,
   inputs: [{name: signer, type: address},{name: amount,
    type: uint128}]
  }
  C1-function_inputs: []
  C2-function_inputs: []
  C3-function_inputs: [
    similar_func: {
      function_name: transfer,
      parameters: [{name: to, type: address}, {name: value,
     type: uint256}]
    },
    inputs:[{0x4f1b06b5f0bddce0d5556dcd8-16b747a6a7de28,
     3000000000000000000}, ...]
    }]
  ]
 }
]
```

Fig. 8. Example of generated C3 inputs for testing a function.

consider the practical inputs of the top-k most similar functions as the recommended test inputs for a given function of a smart contract.

IV. EXPERIMENTAL SETTING

Empirical evaluation of SMARTGIFT is performed through several experiments. Before describing the experimental results, we present the research questions, the data collection, and the overall experimental setup.

A. Research Questions

Our investigation into the performance of generating testing inputs for smart contract functions with SMARTGIFT seeks to answer the following research questions (RQs):

- **RQ1.** *Can* SMARTGIFT *effectively generate practical inputs for executing new smart contracts by learning from the transaction records of smart contracts deployed on Ethereum?* We mainly aim to assess to what extent the hypothesis that relevant inputs for the execution of smart contract functions can be found from their similar counterparts is valid. This RQ allows implicitly to assess the relevance of the proposed embedded function signatures.
- **RQ2.** *To what extent inputs generated by* SMARTGIFT *can be executed successfully?* With this RQ we specifically assess the relevance of executing the target smart contracts with the generated inputs.
- **RQ3.** *Can the inputs generated by* SMARTGIFT *help effectively discover vulnerabilities in smart contracts?* Testing is a widely adopted means to discover vulnerabilities, and various approaches have been proposed in the literature to generate test inputs (e.g., fuzzing and mutation). Therefore, we investigate whether the state of the art in vulnerability detection is improved when leveraging the inputs generated by SMARTGIFT.

B. Data Collection

To answer the aforementioned research questions, we consider inputs for executing smart contracts from the transaction records of smart contracts deployed on Ethereum, a global, open-source platform for decentralized applications.

Etherscan allows to explore and search the smart contracts transactions, addresses, tokens, prices and other activities taking place on Ethereum. With the support of its open API (i.e., `eth_getBlockByNumber`), we can collect the block information from Block #10132253 and a total of 5,000 blocks are collected. We then randomly select 1,000 blocks to extract the related transaction records. Finally, 89,612 transaction records are collected. Such transaction records are presented as binary input data. To identify the function information (i.e., function signature) and the input data for testing at the source code level for each transaction, we leverage the ethereum input data decoder[11] to decode each transaction record. Eventually, 66,528 transactions are successfully resolved with their function signatures and practical inputs.

C. Experimental Setup

To evaluate the performance of our approach, we implement a learning pipeline into a prototype tool also named SMARTGIFT as the approach. In the phase of representation embedding, we use BERT [40] as the deep representation learning model to embed function signatures and generate the related representation numeric vectors. BERT is a deep representation learning model for languages and has been widely used in the domain of software engineering. For example, Zhou et al. [42] applied the BERT model to extract semantic features from code identifiers of programs to perform code recommendation. Yu et al. [43] leveraged BERT on binary code to identify similar binaries. Tian et al. [41] used the BERT model to the semantic similarity between the buggy code and fixed code to identify the correctness of patches generated program repair tools. In the experimental scenario of SMARTGIFT, the *bert_multi_cased_L-12_H-768_A-12*[12] version of BERT is used to embed functions signatures for smart contracts. All our experiments are carried out on a server machine of Ubuntu 18.04 operating system with 32GB memory and 4 cores(Intel i5-9400 CPU, 2.90 GHz).

V. EXPERIMENTAL RESULTS

We now present the experiments that we designed to answer each research question in this study.

A. RQ1: Generation of Practical Inputs with SMARTGIFT

We investigate to what extent practical inputs can be generated for smart contracts by exploring similar functions with SMARTGIFT. To this end, the collected 66,528 transaction records are randomly divided into two groups: a learning dataset (90%, 59 876 cases) of SMARTGIFT and a test dataset (10%, 6 652 cases). For each smart contract sc in the test set, an input $input_r$ generated by SMARTGIFT is considered to be a correct input when it is identical to the original input of sc (excluding its address type). The address is the transaction object of the current smart contract, which can be an account or another smart contract. When testing a smart contract before it is put in production, the address can be a random value since it is not an important parameter for analyzing the actual behaviour of the smart contract.

[11]https://github.com/miguelmota/ethereum-input-data-decoder

[12]https://tfhub.dev/google/bert_multi_cased_L-12_H-768_A-12/

TABLE III
STATISTICAL RESULTS OF GENERATING PRACTICAL INPUTS.

Category	Top-1		Top-5		Top-10	
	# cases	Ratio	# cases	Ratio	# cases	Ratio
C1	3,828	57.5%	3,837	57.7%	3,848	57.8%
C2	69	1.0%	90	1.4%	92	1.4%
C3	0	0	39	0.6%	39	0.6%

Table III presents the statistical results of our experiments. Overall, the inputs of most smart contract functions (∼57.5%) can be successfully generated by SMARTGIFT when restricting the search space on the first category with the top-1 most similar function. When increasing the search space to top-5 and top-10 most similar functions, only a few more inputs can be successfully recommended (very low increasing margin points, i.e, around 0.1%). If the first category does not contain enough inputs (i.e., 10 inputs in this work) for the smart contracts in the test set, the second and third categories can help generate inputs for some cases. However, once again enlarging the search space to top-5 and top-10 only improves marginally the performance of generating practical inputs. Note that, our search for inputs prioritizes the C1-similar functions over C2/C3-similar ones. If a function can be matched with C1-similar functions, C2/C3-similar functions will be discarded. Therefore, our evaluation does not always assess the inputs of C2/C3-similar functions.

[RQ1]: SMARTGIFT *can effectively generate practical inputs for real-world smart contracts functions by learning from the transaction records of real-world smart contracts with strict constraints on parameter data types of functions. However, enlarging the search space with similarities (with Top-5 & Top-10) does not substantially improve the performance of generating practical inputs. Interestingly, relaxing the constraints on parameter data types (with C2 & C3) has the potential to complement the input generation with the first category.*

B. RQ2: Test Relevance of SMARTGIFT-generated Inputs

Automatically generating test inputs aims at boosting the validation of the program by reducing human intervention. In this section, we further assess to what extent inputs generated by SMARTGIFT are indeed relevant to enable the successful execution of the target smart contracts. To this end, we leverage *ExecuWatch* [44], an open-source tool that records the execution details (including the unsuccessfully executed results) of smart contracts by executing test cases.

ExecuWatch provides a default test input generation with fuzzing, which allows to compare the relevance (for successful execution of test cases) of the test inputs generated by SMARTGIFT vs the test inputs generated by the fuzzing method. In this experimental scenario, all of the collected 66,528 real-world smart contract transactions are used as the training data for input generation with SMARTGIFT. As the test set, we use the smart contracts proposed in the *ExecuWatch* paper

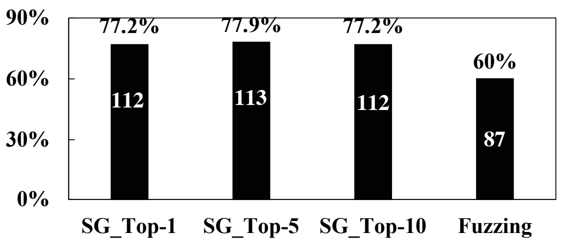

Fig. 9. Comparison on the successfully executed functions, where "SG" stands for SMARTGIFT (the same as Figure 11).

(i.e., 100 smart contracts collected from real world) to avoid potential bias from the inconsistent test dataset. Note that, for assessing the test inputs generated by SMARTGIFT, the original fuzzing process in *ExecuWatch* will be disabled. At the beginning of this experiment, we note that *ExecuWatch* can successfully test 60 smart contracts accounting for 188 functions. Moreover, 43 functions do not have any input parameters. Eventually, 145 (=188-43) functions are selected as the subjects of this experiment. In this experiment, each of the 145 functions is tested 10 times (i.e., 10 groups of testing inputs).

As presented before, SMARTGIFT generates test inputs by considering the similarities between functions. In this scenario, like in RQ1, we also consider the top-1, top-5 and top-10 most similar functions to generate test inputs, respectively. For each of the top-1, top-5 and top-10 cases, ten test inputs are randomly selected to be fed to *ExecuWatch*. Note that, we prioritize the similar functions in the first category (i.e., **C1**) over the other two categories. If the test inputs in the first category are less than 10, SMARTGIFT will generate the test inputs from the second and third categories.

Figure 9 illustrates the statistical results about the functions that are successfully tested with the inputs generated by SMARTGIFT and fuzzing approach, respectively. Overall, the test inputs generated by SMARTGIFT can be used to successfully test 77.2% (=112/145), 77.9% (=113/145) and 77.2% (=112/145) functions in the top-1, top-5 and top-10 cases, respectively, which achieves a higher successful ratio with $\sim$17 percentage points than the fuzzing approach. Note that, we aim to assess whether the test inputs generated by SMARTGIFT are the expected inputs for the test set or not. Thus, we only consider the successfully executed test inputs as the "relevant" test inputs to assess the relevance of SMARTGIFT-generated inputs.

We further investigate the complementary situation of functions that are successfully tested by SMARTGIFT and the fuzzing approach, of which results are presented in Figure 10. The three cases (i.e., top-1, top-5 and top-10) of SMARTGIFT have 110 (=36 + 74) identical functions. Only one or two functions are complementary to each other. Comparing against the fuzzing approach, 10 functions can be tested with fuzzed test inputs but cannot be tested with SMARTGIFT's results. On the other hand, over 37 (= 36 + 1) functions cannot be tested with fuzzed test inputs but can be tested with SMARTGIFT's results. SMARTGIFT can successfully complement fuzz testing (and vice-versa) in the generation of relevant test inputs).

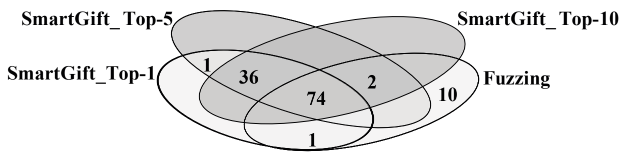

Fig. 10. Complementary situation of the successfully executed functions.

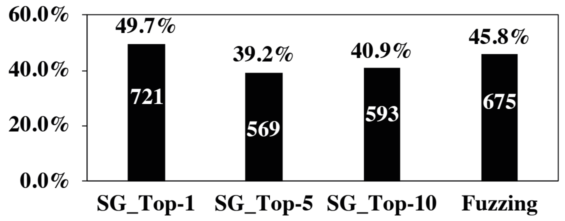

Fig. 11. Comparison on the successfully executed test inputs.

Each of the 145 functions is tested with ten inputs, thus 1,450 trials are proceeded in each of the four cases (i.e., SG_Top-1, SG_Top-5, SG_Top-10, and Fuzzing). We investigate to what extent these trials are successful tests. The statistical results are illustrated in Figure 11. Overall, the succeeded ratios of the four cases are less than 50%. In the top-1 case of SMARTGIFT, 712 tests are successfully executed, which outperforms the fuzzing approach as well as its top-5 and top-10 cases. It indicates that extending the search space of the generated test inputs by considering less similar functions does not effectively improve the performance on finding more relevant test inputs but can reduce such performance.

[RQ2]: SMARTGIFT *can generate relevant inputs for testing real-world smart contracts, which can be a complementary resolution for the fuzzing approach.*

C. RQ3: Detecting Vulnerabilities with SMARTGIFT*-generated Inputs*

In the domain of smart contracts, testing has already been leveraged to uncover vulnerabilities. In this experiment, we assess the possibility of detecting vulnerabilities in smart contracts using the practical inputs generated by SMARTGIFT. To this end, we feed the practical inputs generated by SMARTGIFT to a state-of-the-art vulnerability detector, ContractFuzzer [34], which, by default, relies on a fuzzing method to generate test inputs. In this scenario, SMARTGIFT generates test inputs only relying on the top-1 most similar functions. Note that, when evaluating SMARTGIFT, we disable the fuzzing-based test input generation of ContractFuzzer. Experiments target a benchmark released with ContractFuzzer and which includes a dataset of 459 vulnerable smart contracts. We are able to successfully deploy 154 of those smart contracts and will thus rely on them for this experiment. The remaining 305 vulnerable contracts in the dataset of ContractFuzzer are not provided with the required configuration files, we failed to deploy them.

As shown in Table IV, when ContractFuzzer is fed with test inputs generated by Fuzzing, 136 vulnerabilities can be detected. When ContractFuzzer is fed with test inputs

TABLE IV
VULNERABILITIES DETECTED BY CONTRACTFUZZER WITH FUZZING/SMARTGIFT.

Vulnerability Type	# of Vulnerabilities	Fuzzing	SMARTGIFT
Time Dependency	77	74	71
Number Dependency	25	22	19
Gasless Send	16	13	13
Exception Disorder	25	19	21
Delegate Dangerous	5	5	3
Reentrancy	6	3	4
Total	154	136	131

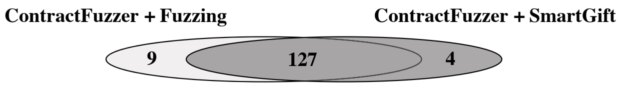

Fig. 12. Comparison between ContractFuzzer with Fuzzing and SMARTGIFT.

generated by SMARTGIFT, 131 vulnerabilities can be detected, and 4 of them (shown in Figure 12) cannot be detected with the test inputs generated with Fuzzing. These results indicate that the test inputs generated by SMARTGIFT can be used to detect the vulnerabilities in smart contracts, and SMARTGIFT can be complementary for the state-of-the-art fuzzing approach to generate test inputs for detecting vulnerabilities in smart contracts.

For the fuzzing approach, it randomly generates the test inputs from a big search space. For example, when the fuzzing approach generates inputs for the integer parameters, the smallest search space is 256 (i.e., 2^8 for `uint8`), while the search space is sharply increased to 2^{256} for `uint256`. With SMARTGIFT, the experimental results show that the smallest search space is only one and the biggest one is 45,302 (while the second biggest search space is just 926). It indicates that SMARTGIFT can sharply reduce the search space for testing smart contracts to improve efficiency. However, SMARTGIFT relies on the practical transactions of real-world smart contracts, which comes to a limitation of generating a few or zero test input for the specific smart contracts. It further leads to the Contract Fuzzer with SMARTGIFT failing to find the vulnerabilities in those contracts. For example, SMARTGIFT failed to generate test inputs for contracts *DSProxy*, *Videos*, *CryptoPoosToken*, *Videos*, and *HashToken*. It is the main reason why 9 vulnerabilities can be detected by ContractFuzzer with Fuzzing but cannot be detected with SMARTGIFT. To address this limitation, the transaction records from more kinds of smart contracts should be collected as the learning data for SMARTGIFT, which is considered as a part of our future work.

[RQ3]: *The practical test inputs generated by* SMARTGIFT *can be used to detect the vulnerabilities in smart contracts, and present the potential of detecting the vulnerabilities that cannot be detected by the fuzzing approach.* SMARTGIFT *can be a complementary resolution for the fuzzing approach to detecting vulnerabilities in smart contracts.*

D. Discovering Potential Issues in Smart Contracts

In the RQ2 experiment, we observed that the success ratios of testing some functions in smart contracts are much lower than others. Figure 13 shows such function `getHolder` that is executed with the inputs generated by SMARTGIFT for a low successful ratio.

```
contract Distrubution {
   ...
   function getHolder(uint _number) external view returns(
     address) {
       return holders[_number];
   }
   ...
}
```

Fig. 13. One function case with low successful rate.

After looking at the code, we note that the function `getHolder` is to get one element from the array `holders` that saves several account addresses. However, the developer does not provide any conditional assert to check the value range of the parameter `_number`. If the value of `_number` is larger than the real length of the array `holders`, the execution of the transaction will fail because of the overflow exception. It is difficult to guarantee that the practical inputs will satisfy the length of the array `holders`. To address the potential issues and improve the robustness of the smart contract, developers should add the checking for the range of the parameter value. This is not uncovered by the original test inputs of the contract, it implies that SMARTGIFT could be capable of generating the testing inputs for uncovering the issues of corner cases.

VI. THREATS TO VALIDITY

The external threats to validity include the scale of the dataset collected from the Etherscan. In this paper, we only collect 66,528 transaction records because of our limitations in the crawling process. We consider enlarging the dataset as a part of our future work. The collected transaction records are decoded into the original values by the *ethereum-input-data-decoder*. This decoder is challenged in decoding some input types like *bytes* and *array*, which could generate inconsistent data values for parameters and threaten the practical input generation with SMARTGIFT. We rely on the function signatures to find similar functions for the given functions. However, some functions could be named with identifiers that are inconsistent with their concrete implementation (cf. related work on method naming inconsistencies by Liu et al. [39]). To alleviate this threat, it would be necessary to check the consistency between the function names and implementation for collecting a more reliable dataset.

The internal threats to validity stem from the deep representation learning of function signatures: SMARTGIFT relies on pre-trained BERT model, which is actually better adapted to natural language, rather than code. To address this limitation, we are planning to re-train a deep representation learning model for the similarity of functions in smart contracts, as well as consider the context of the smart contracts and the concrete implementation of each function. Two functions have different names but their bodies may be similar at the AST level, such similar functions are not covered by our method, So we plan for future work to consider semantic information from function implementations (e.g., AST) to further improve the selection of adequate test inputs. The internal threats to validity could include the practical validation of generated test cases. To solve this threat, it would be more effective to apply questionnaire surveys or interviews with professional smart contract developers, we plan it as the future work to prove the practicability of SmartGift.

VII. Related Work

Testing for Smart Contracts. Smart contracts are computer programs written in a domain-specific language (such as Solidity). These programs face the same challenge as the traditional software programs with issues (i.e., bugs and vulnerabilities) that should be uncovered and fixed before they are deployed. Vulnerabilities in smart contracts are further critical since they are immutable once deployed. To improve the robustness of smart contract programs, various testing approaches [25], [29], [30], [31], [33]. have been proposed in the literature for detecting the vulnerabilities in smart contracts.

Jiang et al. [34] proposed to build seed inputs with the valid input domain and the inputs frequently used by some types in smart contracts, to fuzz inputs for testing smart contracts. ReGuard [35] leveraged the fuzz testing for smart contracts by iteratively generating random but diverse transactions to detect reentrancy vulnerabilities in smart contracts. Ma et al. [36] combined the feedback-directed mutation with the parameter data types of functions to fuzz test inputs for smart contracts. Honig et al. [28] proposed a mutation testing framework for the Solidity contracts deployed on the Ethereum blockchain. Harvey [27] and sFuzz [32] are proposed by leveraging the fuzzing approach to discover the vulnerabilities for smart contracts as well. According to analyzing the Solidity documents and issues about smart contracts listed in GitHub and Stack Exchange, Liu et al. [37] defined 15 novel mutation operators for generating inputs to test smart contracts. Inspiring from the real faults in Solidity smart contracts, Andersta et al. [38] designed 10 mutation operators for generating test inputs.

The state-of-the-art test case generation for smart contracts, however, is highly dependent on the quality of existing inputs, of which generated test inputs are associated with high randomness, that could impact the effectiveness of generated inputs. In addition, the test inputs provided in test suites of smart contracts are always impractical, which are set with simple values or randomly provided by developers. Different from the existing research work, according to mining the similarities between functions in smart contracts, SmartGift is to generate the practical test inputs by learning from the actionable inputs of functions that are encoded in the transaction records for executing real-world smart contracts.

Deep Representation Learning for Programs. In recent years, deep representation learning techniques have been widely studied in the domain of program code. Liu et al. [45] leveraged the convolutional neural networks to address the pattern mining task. Soto and Le Goues [46] built a probabilistic model to predict bug fixes for program repair. Hoang et al. [47] proposed a hierarchical deep learning- based method with features extracted from both commit messages and commit code to identify stable Linux patches. Tian et al. [41] leverage the representation learning techniques to evaluate the correctness of patches for buggy programs. Liu et al. [39] and Allamanis et al. [48], [49] explored the consistent relationship between the identifiers and the related concrete code implementation.

Godefroid et al. [50] used a learned input probability distribution to intelligently guide where to fuzz inputs. Zong et al. [51] proposed a deep-learning-based approach to predict the reachability of inputs and filter out the unreachable ones to boost the performance of fuzzing. Wang et al. [52] leveraged the knowledge in the vast amount of existing samples to learn a probabilistic context-sensitive grammar to generate seed inputs for fuzz testing. Our work aims at leveraging the representation learning technique to embed function signatures of smart contracts to contribute to the generation of the practical test inputs.

VIII. Conclusion

This paper contributes to the community effort on developing approaches and tool support for efficiently testing smart contracts. Our focus has been to improve the generation of inputs that would be most suitable for uncovering vulnerabilities. We propose to that end the SmartGift approach, which learns from transaction records of real-world smart contracts in order to generate practical test inputs for new smart contracts under test. Our prototype implementation leverages the BERT pre-trained model to embed the signatures of functions allowing an efficient search of similar functions from a large dataset of functions implemented in real-world deployed smart contracts. With 66,528 collected transaction records, SmartGift was able to generate relevant test inputs for $\sim$77% smart contract functions, outperforming a baseline fuzzing approach. We further show that the inputs generated by SmartGift are complementary to those obtained through fuzzing. We further assess the potential of detecting vulnerabilities in smart contracts with the test inputs generated by SmartGift. By feeding ContractFuzzer, a state of the art tool, with test inputs generated by SmartGift, 131 of 154 smart contract vulnerabilities from a benchmark could be detected correctly. These experimental results show that the practical inputs generated by SmartGift are meaningful to test smart contracts and present the potential of detecting vulnerabilities in smart contracts.

Acknowledgements

This research was supported by the National Key R&D Program of China (No. 2020YFB1005500 and 2020AAA010 7704), the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06), and the Open Project Program of the Key Laboratory of Safety-Critical Software (NUAA), Ministry of Industry and Information Technology (No. XCA20026), as well as the funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949014).

References

[1] M. Swan, *Blockchain: Blueprint for a new economy*. O'Reilly Media, Inc., 2015.

[2] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[3] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies*. O'Reilly Media, Inc., 2014.

[4] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[5] J. Reed, "Litecoin: An introduction to litecoin cryptocurrency and litecoin mining," 2017.

[6] M. Alharby and A. van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *CoRR*, vol. abs/1710.06372, 2017. [Online]. Available: http://arxiv.org/abs/1710.06372

[7] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, 2019.

[8] N. Szabo, "Smart Contracts," https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html, 1994.

[9] P. Hegedus, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," in *Proceedings of the IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. IEEE, 2018, pp. 35–39.

[10] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," in *Proceedings of the International conference on financial cryptography and data security*. Springer, 2017, pp. 494–509.

[11] M. del Castillo, "The DAO attacked: Code issue leads to \$60 million ether theft," 2016. [Online]. Available: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft

[12] R.-R. O'Leary, "Parity team publishes postmortem on \$160 million ether freeze," 2017. [Online]. Available: https://www.coindesk.com/parity-team-publishes-postmortem-160-million-ether-freeze

[13] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[14] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 814–819. [Online]. Available: https://doi.org/10.1145/3238147.3240728

[15] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 664–676. [Online]. Available: https://doi.org/10.1145/3274694.3274737

[16] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "EthIR: A framework for high-level analysis of ethereum bytecode," in *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, vol. 11138. Springer, 2018, pp. 513–520. [Online]. Available: https://doi.org/10.1007/978-3-030-01090-4_30

[17] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[18] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "SAFEVM: a safety verifier for ethereum smart contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 386–389. [Online]. Available: https://doi.org/10.1145/3293882.3338999

[19] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, 2019, pp. 1591–1607. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira

[20] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. C. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8979435

[21] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *Proceedings of thee 29th USENIX Security Symposium*. USENIX Association, 2020, pp. 2757–2774. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/frank

[22] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. T. Vechev, "VerX: Safety verification of smart contracts," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1661–1677. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00024

[23] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VERISMART: A highly precise safety verifier for ethereum smart contracts," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1678–1694. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00032

[24] P. Zhang, F. Xiao, and X. Luo, "Soliditycheck : Quickly detecting smart contract problems through regular expressions," *CoRR*, vol. abs/1911.09425, 2019. [Online]. Available: http://arxiv.org/abs/1911.09425

[25] P. Zhang, J. Yu, and S. Ji, "ADF-GA: data flow criterion based test case generation for ethereum smart contracts," *CoRR*, vol. abs/2003.00257, 2020. [Online]. Available: https://arxiv.org/abs/2003.00257

[26] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "SMARTSHIELD: automatic smart contract protection made easy," in *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2020, pp. 23–34. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054825

[27] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," *CoRR*, vol. abs/1905.06944, 2019. [Online]. Available: http://arxiv.org/abs/1905.06944

[28] J. J. Honig, M. H. Everts, and M. Huisman, "Practical mutation testing for smart contracts," in *Proceedings of the 2019 International Workshops on Data Privacy Management, Cryptocurrencies and Blockchain Technology*, ser. Lecture Notes in Computer Science, vol. 11737. Springer, 2019, pp. 289–303. [Online]. Available: https://doi.org/10.1007/978-3-030-31500-9_19

[29] Y. Huang, B. Jiang, and W. K. Chan, "EOSFuzze: Fuzzing EOSIO smart contracts for vulnerability detection," *CoRR*, vol. abs/2007.14903, 2020. [Online]. Available: https://arxiv.org/abs/2007.14903

[30] Q. Zhang, Y. Wang, J. Li, and S. Ma, "EthPloit: From fuzzing to efficient exploit generation against smart contracts," in *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2020, pp. 116–126. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054822

[31] P. H. Hartel and R. Schumi, "Mutation testing of smart contracts at scale," in *Proceedings of the 14th International Conference on Tests and Proofs*, ser. Lecture Notes in Computer Science, vol. 12165. Springer, 2020, pp. 23–42. [Online]. Available: https://doi.org/10.1007/978-3-030-50995-8_2

[32] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: an efficient adaptive fuzzer for solidity smart contracts," *CoRR*, vol. abs/2004.08563, 2020. [Online]. Available: https://arxiv.org/abs/2004.08563

[33] I. Ashraf, X. Ma, B. Jiang, and W. K. Chan, "GasFuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities," *IEEE Access*, vol. 8, pp. 99 552–99 564, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.2995183

[34] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269. [Online]. Available: https://doi.org/10.1145/3238147.3238177

[35] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 65–68. [Online]. Available: https://doi.org/10.1145/3183440.3183495

[36] F. Ma, Y. Fu, M. Ren, W. Sun, Z. Liu, Y. Jiang, J. Sun, and J. Sun, "GasFuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability," *CoRR*, vol. abs/1910.02945, 2019. [Online]. Available: http://arxiv.org/abs/1910.02945

[37] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen, "MuSC: A tool for mutation testing of ethereum smart contract," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software*

Engineering. IEEE, 2019, pp. 1198–1201. [Online]. Available: https://doi.org/10.1109/ASE.2019.00136

[38] E. Andesta, F. Faghih, and M. Fooladgar, "Testing smart contracts gets smarter," *CoRR*, vol. abs/1912.04780, 2019. [Online]. Available: http://arxiv.org/abs/1912.04780

[39] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. L. Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 2019, pp. 1–12. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00019

[40] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[41] H. Tian, K. Liu, A. K. Kaboreé, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proceedings of 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[42] S. Zhou, B. Shen, and H. Zhong, "Lancer: Your code tell me what you need," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2019, pp. 1202–1205. [Online]. Available: https://doi.org/10.1109/ASE.2019.00137

[43] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the 34th AAAI Conference on Artificial Intelligence*. AAAI Press, 2020, pp. 1145–1152. [Online]. Available: https://aaai.org/ojs/index.php/AAAI/article/view/5466

[44] D. Wang, K. Liu, and L. Li, "On the need of understanding the failures of smart contracts," *IEEE Software*, vol. 37, no. 4, pp. 49–54, 2020. [Online]. Available: https://doi.org/10.1109/MS.2020.3003921

[45] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, 2018.

[46] M. Soto and C. Le Goues, "Using a probabilistic model to predict bug fixes," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 221–231.

[47] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "PatchNet: hierarchical deep learning-based stable patch identification for the linux kernel," *CoRR*, vol. abs/1911.03576, 2019. [Online]. Available: http://arxiv.org/abs/1911.03576

[48] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[49] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning*. JMLR.org, 2016, pp. 2091–2100.

[50] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2017, pp. 50–59. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115618

[51] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proceedings of the 29th USENIX Security Symposium*. USENIX Association, 2020, pp. 2255–2269. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zong

[52] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. IEEE, 2017, pp. 579–594. [Online]. Available: https://doi.org/10.1109/SP.2017.23