This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3182373

xFuzz: Machine Learning Guided Cross-Contract Fuzzing

Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao

Abstract—Smart contract transactions are increasingly interleaved by cross-contract calls. While many tools have been developed to identify a common set of vulnerabilities, the cross-contract vulnerability is overlooked by existing tools. Cross-contract vulnerabilities are exploitable bugs that manifest in the presence of more than two interacting contracts. Existing methods are however limited to analyze a maximum of two contracts at the same time. Detecting cross-contract vulnerabilities is highly non-trivial. With multiple interacting contracts, the search space is much larger than that of a single contract. To address this problem, we present XFUZZ, a machine learning guided smart contract fuzzing framework. The machine learning models are trained with novel features (e.g., word vectors and instructions) and are used to filter likely benign program paths. Comparing with existing static tools, machine learning model is proven to be more robust, avoiding directly adopting manually-defined rules in specific tools. We compare XFUZZ with three state-of-the-art tools on 7,391 contracts. XFUZZ detects 18 exploitable cross-contract vulnerabilities, of which 15 vulnerabilities are exposed for the first time. Furthermore, our approach is shown to be efficient in detecting non-cross-contract vulnerabilities as well—using less than 20% time as that of other fuzzing tools, XFUZZ detects twice as many vulnerabilities.

Index Terms—Smart Contract, Fuzzing, Cross-contract Vulnerability, Machine Learning

1 INTRODUCTION

ETHEREUM has been on the forefront of most rankings of block-chain platforms in recent years [1]. It enables the execution of programs, called smart contracts, written in Turing-complete languages such as Solidity. Smart contracts are increasingly receiving more attention, e.g., with over 1 million transactions per day since 2018 [2].

At the same time, smart contracts related security attacks are on the rise as well. According to [3], [4], [5], vulnerabilities in smart contracts have already led to devastating financial losses over the past few years. In 2016, the notorious DAO attack resulted in the loss of 150 million dollars [6]. Additionally, as figured out by Zou *et al.* [7], over 75% of developers agree that the smart contract software has a much high security requirement than traditional software. Considering the close connection between smart contract and financial activities, the security of smart contract security largely effects the stability of the society.

Many methods and tools have since been developed to analyze smart contracts. Existing tools can roughly be categorized into two groups: *static analyzers* and *dynamic analyzers*. Static analyzers (e.g., [8], [9], [10], [11], [12], [13]) often leverage static program analysis techniques (e.g., symbolic execution and abstract interpretation) to identify suspicious program traces. Due to the well-known limitations of static analysis, there are often many false alarms. On the other side, dynamic analyzers (including fuzzing engines such as [14], [15], [16], [17], [18]) avoid false alarms by dynamically executing the traces. Their limitation is that there can often be a huge number of program traces to execute and thus smart strategies must be developed to selectively test the program traces in order to identify as many vulnerabilities as possible. Besides, static and dynamic tools also have a common drawback — *the detection rules are usually built-in and predefined by developers*, sometimes the rules among different tools could be contradictory (e.g., reentrancy detection rules in SLITHER and OYENTE [19]).

While existing efforts have identified an impressive list of vulnerabilities, one important category of vulnerabilities, i.e., cross-contract vulnerabilities, has been largely overlooked so far. Cross-contract vulnerabilities are exploitable bugs that manifest only in the presence of more than two interacting contracts. For instance, the reentrancy vulnerability shown in Figure 4 occurs only if three contracts interact in a particular order. In our preliminary experiment, the two well-known fuzzing engines for smart contracts, i.e., CONTRACTFUZZER [15] (version 1.0) and SFUZZ [14] (version 1.0), both missed this vulnerability due to they are limited to analyze two contracts at the same time.

Given a large number of cross-contract transactions in practice [20], there is an urgent need for developing systematic approaches to identify cross-contract vulnerabilities. Detecting cross-contract vulnerabilities however is non-trivial. With multiple contracts involved, the search space is much larger than that of a single contract, i.e., we must

- *Yinxing Xue and Wei Zhang are with the University of Science and Technology of China. E-mail: yxxue@ustc.edu.cn, sa190@mail.ustc.edu.cn.*
- *Jiaming Ye and Jianjun Zhao are with the Kyushu University. Email: ye.jiaming.852@s.kyushu-u.ac.jp, zhao@ait.kyushu-u.ac.jp.*
- *Jun Sun is with the Singapore Management University. E-mail: junsun@smu.edu.sg.*
- *Lei Ma is with the University of Alberta. E-mail: ma.lei@acm.org.*
- *Haijun Wang is with the Nanyang Technological University. E-mail: hjwang.china@gmail.com.*

Manuscript received December 22, 2021; revised April 14, 2022; accepted June 2, 2022. Date of publication July 2, 2022; date of current version June 5, 2022. This work was supported in part by National Nature Science Foundation of China under Grant 61972373, in part by the Basic Research Program of Jiangsu Province under Grant BK20201192 and in part by the National Research Foundation Singapore under its NSoE Programme (Award Number: NSOE-TSS2019-03). The research of Dr Xue is also supported by CAS Pioneer Hundred Talents Program of China. (Yinxing Xue and Jiaming Ye are co-first authors. Yinxing Xue is the corresponding author).

consider all sequences and interleaving of function calls from multiple contracts.

As fuzzing techniques practically run programs and barely produce false positive reports [15], [21], adopting fuzzing in cross-contract vulnerability detection is preferred. However, due to the efficiency concerns, we need other techniques to guide fuzzers to practically detect cross-contract vulnerabilities. Previous works (e.g., [22], [23]) have evidenced the advantages of applying machine learning method for improving efficiency of vulnerability fuzzing in C/C++ programs. Compared with static rule-based methods, the ML model based method requires no prior domain knowledge about known vulnerabilities, and can effectively reduce the large search space for covering more vulnerable functions. In smart contract, existing works (e.g., ILF [24]) focus on exploring the state space in the *intra-contract* scope. They are unable to address the cross-contract vulnerabilities. With a large search space of combinations of numerous function calls, it is desired to guide the fuzzing process via the aid of the machine learning models.

In this work, we propose XFUZZ, a machine learning (ML) guided fuzzing engine designed for detecting cross-contract vulnerabilities. Ideally, according to the Pareto principle in testing [25] (i.e., roughly 80% of errors come from 20% of the code), *we want to rapidly identify the error-prone code before applying the fuzzing technique*. As reported by previous works [26], [27], the existing analysis tools suffer from high false positive rates (e.g., SLITHER [10] and SMARTCHECK [13] have more than 70% of false positive rates). Therefore, adopting only one static tool in our approach may produce biased results. To alleviate this, we use three tools to vote the reported vulnerabilities in contracts, and we further train a ML model to learn common patterns from the voting results. It is known that ML models can automatically learn patterns from inputs with less bias [28]. Based on this, the overall bias due to using a certain tool to identify potentially vulnerable functions in contracts can be reduced.

Specifically, XFUZZ provides multiple ways of reducing the enormous search space. First, XFUZZ is designed to leverage an ML model for identifying the most probably vulnerable functions. That is, an ML model is trained to filter most of the benign functions whilst preserving most of the vulnerable functions. During the training phase, the ML models are trained based on a training dataset that contains program codes that are labeled using three famous static analysis tools (i.e., the labels are their majority voting result). Furthermore, the program code is vectorized into vectors based on word2vec [29]. In addition, manually designed features, such as `can_send_eth`, `has_call` and `callee_external`, are supplied to improve training effectiveness as well. In the guided fuzzing phase, the model is used to predict whether a function is potentially vulnerable or not. In our evaluation of ML models, the models allow us to filter 80.1% non-vulnerable contracts. Second, to further reduce the effort required to expose cross-contract vulnerabilities, the filtered contracts and functions are further prioritized based on a suspiciousness score, which is defined based on an efficient measurement of the likelihood of covering the program paths.

To validate the usefulness of XFUZZ, we performed comprehensive experiments, comparing with a static cross-contract detector CLAIRVOYANCE [19] and two state-of-the-art dynamic analyzers, i.e., CONTRACTFUZZER [15] and SFUZZ, on widely-used open-dataset ([30], [31]) and additional 7,391 contracts. The results confirm the effectiveness of XFUZZ in detecting cross-contract vulnerabilities, i.e., 18 cross-contract vulnerabilities have been identified. 15 of them are missed by all the tested state-of-the-art tools. We also show that our search space reduction and prioritization techniques achieve high precision and recall. Furthermore, our techniques can be applied to improve the efficiency of detecting intra-contract vulnerabilities, e.g., XFUZZ detects twice as many vulnerabilities as that of SFUZZ and uses less than 20% of time.

The contributions of this work are summarized as follows.

- To the best of our knowledge, we make the first attempts to formulate and detect three common *cross-contract* vulnerabilities, i.e., reentrancy, delegatecall and tx-origin.
- We propose a novel ML based approach to significantly reduce the search space for exploitable paths, achieving well-trained ML models with a recall of 95% on a testing dataset of 100K contracts. We also find that the trained model can cover a majority of reports of other tools.
- We perform a large-scale evaluation and performed comparative studies with state-of-the-art tools. Leveraging the ML models, XFUZZ outperforms the state-of-the-art tools by at least 42.8% in terms of recall meanwhile keeping a satisfactory precision of 96.1%.
- XFUZZ also finds 18 cross-contract vulnerabilities. All of them are verified by security experts from our industry partner. We have published the exploiting code to these vulnerabilities on our anonymous website [32] for public access.

2 MOTIVATION

In this section, we first introduce three common types of cross-contract vulnerabilities. Then, we discuss the challenges in detecting these vulnerabilities by state-of-the-art fuzzing engines to motivate our work.

2.1 Problem Formulation and Definition

In general, smart contracts are compiled into opcodes [33] so that they can run on EVM. We say that a smart contract is *vulnerable* if there exists a program trace that allows an attacker to gain certain benefit (typically financial) illegitimately. Formally, a vulnerability occurs when there exist dependencies from certain critical instructions (e.g., `TXORIGIN` and `DELEGATECALL`) to a set of specific instructions (e.g., `ADD`, `SUB` and `SSTORE`). Therefore, to formulate the problem, we adopt definitions of vulnerabilities from [9], [34], based on which we define (control and data) dependency and then define the cross-contract vulnerabilities.

Definition 1 (Control Dependency). An opcode op_j is said to be control-dependent on op_i if there exists an execution from op_i to op_j such that op_j post-dominates all op_k in the path from op_i to op_j (excluding op_i) but does not post-dominates op_i. An opcode op_j is said to post-dominate an opcode op_i if all traces starting from op_i must go through op_j.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3182373


```
1 function withdrawBalance() public {
2   uint amountToWithdraw = userBalances[msg.
        sender];
3   msg.sender.call.value(amountToWithdraw)("");
4   userBalances[msg.sender] = 0;
5 }
```

Fig. 1: An example of reentrancy vulnerability.

```
1  contract Delegate {
2    address public owner;
3    function pwn() {
4      owner = msg.sender;
5    } }
6  contract Delegation {
7    address public owner;
8    Delegate delegate;
9    function() {
10     if(delegate.delegatecall(msg.data)) {
11       this;
12   } } }
```

Fig. 2: An example of delegatecall vulnerability.

```
1 function withdrawAll(address _recipient) public
      {
2   require(tx.origin == owner);
3   _recipient.transfer(this.balance);
4 }
```

Fig. 3: An example of tx-origin vulnerability.

Definition 2 (Data Dependency). An opcode op_j is said to be data-dependent on op_i if there exists a trace that executes op_i and subsequently op_j such that $W(op_i) \cap R(op_j) \neq \emptyset$, where $R(op_j)$ is a set of locations read by op_j and $W(op_i)$ is a set of locations written by op_i.

An opcode op_j is *dependent* on op_i if op_j is *control or data dependent* to op_i or op_j is dependent to op_k meanwhile op_k is dependent to op_i.

In this work, we define three typical categories of cross-contract vulnerabilities that we focus on, i.e., reentrancy, delegatecall and tx-origin. Although our method can be generalized to support more types of vulnerabilities, in this paper, we focus on the above three vulnerabilities since they are among the most dangerous ones with urgent testing demands. Specifically, the reentrancy and delegatecall vulnerabilities are highlighted as top risky vulnerabilities in previous works [9], [10]. The tx-origin vulnerability is broadly warned in previous research [35], [10].

We define C as a set of critical opcodes, which contains `CALL`, `CALLCODE`, `DELEGATECALL`, i.e., the set of all opcode associated with external calls. These opcodes associated with external calls could be the causes of vulnerabilities (since then the code is under the control of external attackers).

Definition 3 (Reentrancy Vulnerability). A trace suffers from reentrancy vulnerability if it executes an opcode $op_c \in C$ and subsequently executes an opcode op_s in the same function such that op_s is `SSTORE`, and op_c depends on op_s.

A smart contract suffers from reentrancy vulnerability if and only if at least one of its traces suffers from reentrancy vulnerability. This vulnerability results from the incorrect use of external calls, which are exploited to construct a call-chain. When an attacker A calls a user U to withdraw money, the fallback function in contract A is invoked. Then, the malicious fallback function calls back to U to recursively steal money. In Figure 1, the attacker can construct an end-to-end call-chain by calling `withdrawBalance` in the fallback function of the attacker's contract then steals money.

Definition 4 (Dangerous Delegatecall Vulnerability). A trace suffers from dangerous delegatecall vulnerability if it executes an opcode $op_c \in C$ that depends on an opcode `DELEGATECALL`.

A smart contract suffers from delegatecall vulnerability if and only if at least one of its traces suffers from delegatecall vulnerability. This vulnerability is due to the abuse of dangerous opcode `DELEGATECALL`. When a malicious attacker B calls contract A by using `delegatecall`, contract A's function is executed in the context of attacker, and thus causes damages. In Figure 2, malicious attacker B sends ethers to contract `Delegation` to invoke the fallback function at line 10. The fallback function calls contract `Delegate` and executes the malicious call data `msg.data`. Since the call data is executed in the context of `Delegate`, the attacker can change the owner to an arbitrary user by executing `pwn` at line 3.

Definition 5 (Tx-origin Misuse Vulnerability). A trace suffers from tx-origin misuse vulnerability if it executes an opcode $op_c \in C$ that depends on an opcode `ORIGIN`.

A smart contract suffers from tx-origin vulnerability if and only if at least one of its traces suffers from tx-origin vulnerability. This vulnerability is due to the misuse of `tx.origin` to verify access. An example of such vulnerability is shown in Figure 3. When a user U calls a malicious contract A, who intends to forward call to contract B. Contract B relies on vulnerable identity check (i.e., `require(tx.origin == owner)` at line 2 to filter malicious access. Since `tx.orign` returns the address of U (i.e., the address of `owner`), malicious contract A successfully poses as U.

Definition 6 (Cross-contract Vulnerability). A group of contracts suffer from cross-contract vulnerability if there is a vulnerable trace (that suffers from reentrancy, delegatecall, tx-origin) due to opcode from more than two contracts.

A smart contract suffers from cross-contract vulnerability if and only if at least one of its traces suffers from cross-contract vulnerability. For example, a cross-contract reentrancy vulnerability is shown in Figure 4. An attack requires the participation of three contracts: malicious contract `Logging` deployed at `addr_m`, logic contract `Logic` deployed at `addr_l` and wallet contract `Wallet` deployed at `addr_w`. First, the attack function `log` calls function `logging` at `Logic` contract then sends ethers to the attacker contract by calling function `withdraw` at contract `Wallet`. Next, the wallet contract sends ethers to attacker contract and calls function `log`. An end-to-end call chain ① → ② → ③ → ④ → ①... is formed and the attacker can recursively steal money without any limitations.

2.2 State-of-the-arts and Their Limitations

First, we perform an investigation on the capability in detecting vulnerabilities by the state-of-the-art methods, including

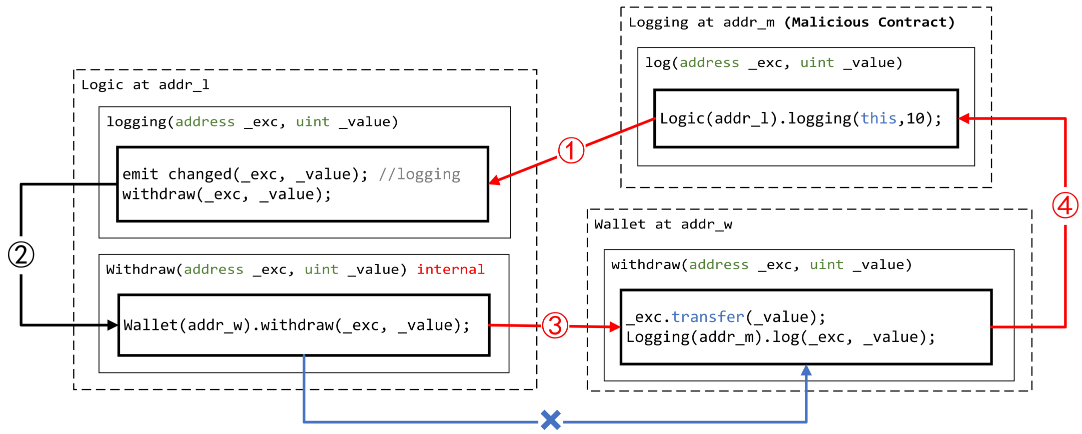


Fig. 4: An example of cross-contract reentrancy vulnerability which is missed by the state-of-art fuzzer, namely sFuzz. *Note: The solid boxes represent functions and the dashed containers denote contracts. Specifically, function call is denoted by solid line. The cross-contract calls are highlighted by red arrows. The blue arrow represents cross-contract call missed by sFuzz and ContractFuzzer.

[10], [8], [9], [19], [14], [15]. In general, cross-contract testing and analysis are not supported by most of these tools except Clairvoyance. The reason is existing approaches merely focus on one or two contracts, and thus, the sequences and interleavings of function calls from multiple contracts are often ignored. For example, the vulnerability in Figure 4 is a false negative case of static analyzer Slither, Oyente and Securify. Note that although this vulnerability is found by Clairvoyance, this tool however generates many false alarms, making the confirmation of which rather difficult. This could be a common problem for many static analyzers.

Although high false positive rate could be well addressed by fuzzing tools by running contracts with generated inputs, existing techniques are limited to maximum two contracts (i.e., input contract and tested contract). In our investigation of two currently representative fuzzing tools sFuzz and ContractFuzzer, cross-contract calls are largely overlooked, and thus leads to missed vulnerabilities. To sum up, most of the existing methods and tools are still limited to handle non-cross-contract vulnerabilities, which motivates this work to bridge such a gap towards solving the currently urgent demands.

3 Overview

Detecting cross-contract vulnerability often requires examining a large number of sequence transactions and thus can be quite computationally expensive some even infeasible. In this section, we give an overall high-level description of our method, e.g., focusing on fuzzing suspicious transactions based on the guideline of a machine learning (ML) model. Technically, there are three challenges of leveraging ML to guide the effective fuzzing cross-contracts for vulnerability detection:

- **C1** How to train the machine learning model and achieve *satisfactory* precision and recall.
- **C2** How to combine trained model with fuzzer to reduce search space towards *efficient* fuzzing.
- **C3** How to empower the guided fuzzer the support of *effective* cross-contract vulnerability detection.

In the rest of this section, we provide an overview of xFuzz which aims at addressing the above challenges, as shown in Figure 5. Generally, the framework can be separated into two phases: *machine learning model training phase* and *guided fuzzing phase*.

3.1 Machine Learning Model Training Phase

In previous works [36], [37], fuzzers are limited to prior knowledge of vulnerabilities and they are not well generalized against vulnerable variants. In this work, we propose to leverage ML predictions to guide fuzzers. The benefit of using ML instead of a particular static tool is that ML model can reduce bias introduced by manually defined detection rules.

In this phase, we collect training data, engineer features, and evaluate models. First, we employ the state-of-the-arts Slither, Securify and Solhint to detect vulnerabilities on the dataset. Next, we collect their reports to label contracts. The contract gains at least two votes are labeled as vulnerability. After that, we engineer features. The input contracts are compiled into bytecode then vectorized into vectors by Word2Vec [29]. To address **C1**, they are enriched by combining with static features (e.g., `can_send_eth`, `has_call` and `callee_external`, etc.). These static features are extracted from ASTs and CFGs. Eventually, the features are used as inputs to train the ML models. In particular, the precision and recall of models are evaluated to choose three candidate models (e.g., XGBoost [38], EasyEnsembleClassifier [39] and Decision Tree), among which we select the best one.

3.2 Guided Testing Phase

In guided testing phase, contracts are input to the pretrained models to obtain predictions. After that, the vulnerable contracts are analyzed and pinpointed. To address challenge **C2**, the functions that are predicted as suspiciously vulnerable ones. Then we use call-graph analysis and control-flow-graph analysis to construct cross-contract call path. After we collect all available paths, we use the path prioritization algorithm to prioritize them. The prioritization becomes the guidance of

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3182373

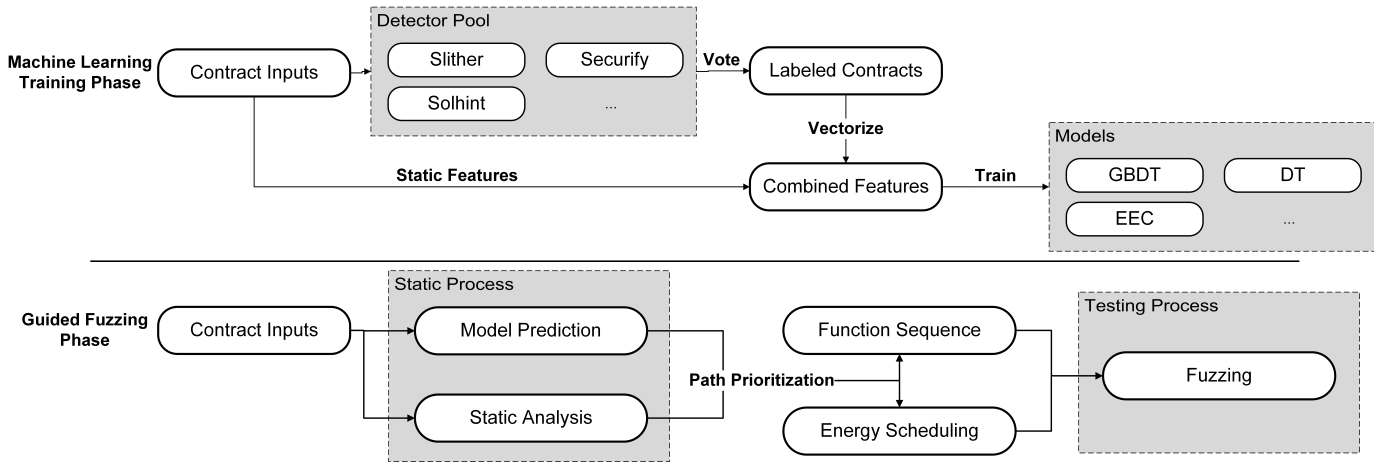


Fig. 5: The overview of XFUZZ framework.

TABLE 1: Vulnerability detection capability of voting static tools.

	Slither	Solhint	Securify
Reentrancy	●	●	●
Tx-origin	●	●	
Delegatecall	●		

the fuzzer. This guidance of model predictions significantly reduces search space because the benign functions wait until the vulnerable ones finish. The fuzzer can focus on vulnerable functions and report more vulnerabilities.

To address **C3**, we extract static information (e.g., function parameters, conditional paths) of contracts to enrich model predictions. The predictions and the static information are combined to compute path priority scores. Based on this, the most exploitable paths are prioritized, where vulnerabilities are more likely found. Here, the search space of exploitable paths is further reduced and the cross-contract fuzzing is therefore feasible by invoking vulnerability through available paths.

4 MACHINE LEARNING GUIDANCE PREPARATION

In this section, we elaborate on the training of our ML model for fuzzing guidance. We discuss the data collection in Section 4.1 and introduce feature engineering in Section 4.2, followed by candidate model evaluation in Section 4.3.

4.1 Data Collection

SMARTBUGS [31] and SWCREGISTRY [40] are two representatives of existing smart contract vulnerability benchmarks. However, their labeled data is scarce and the amount currently available is insufficient to train a good model. Therefore, we choose to download and collect contracts from Etherscan (https://etherscan.io/), a prominent Ethereum service platform. Overall, to be representative, we collect a large set of 100,139 contracts in total for further processing.

The collected dataset is then labeled based on the voting results of three most well-rated static analyzers (i.e., SOLHINT [11] v2.3.1, SLITHER [10] v0.6.9 and SECURIFY [9] v1.0). The three tools are chosen based on the fact that they are ❶ state-of-the-art static analyzers and ❷ well maintained and frequently updated. The detection capability vary among these tools (as shown in Table 1). We then vote to label the dataset aiming at eliminating the bias of each tool. Note that the two vulnerabilities (i.e., delegatecall and tx-origin) are hardly supported by existing tools. Therefore, we only vote vulnerable functions on vulnerabilities supported by at least two tools. That is, for reentrancy, the voting results are counted in the way that the function gain at least two votes is deemed as vulnerability; for tx-origin, the function is deemed as vulnerability when it gains at least one vote. As for delegatecall vulnerability, we label all reported functions as vulnerable ones.

As a result, we collect 788 reentrancy, 40 delegatecall and 334 tx-origin vulnerabilities, respectively. All of the above vulnerabilities are manually confirmed by two authors of this paper, both of whom have more than 3 years development experience for smart contracts, to remove false alarms.

TABLE 2: The seven static features adopted in model training

Feature Name	Type	Description
has_modifier	bool	whether has a modifier
has_call	bool	whether contains a call operation
has_delegate	bool	whether contains a delegatecall
has_tx_origin	bool	whether contains a tx-origin operation
has_balance	bool	whether has a balance check operation
can_send_eth	bool	whether supports sending ethers
callee_external	bool	whether contains external callees

4.2 Feature Engineering

Then, both vulnerable and benign functions are preprocessed by SLITHER to extract their runtime bytecode. After that, Word2Vec [29] is leveraged to transform the bytecode into a 20-dimensional vector. However, as reported in [41], vectors alone are still insufficient for training a high-performance model. To address this, we enrich the vectors with 7 additional static features extracted from CFGs. In short, the features are 27 dimensions in total, in which 20 are yielded by Word2Vec and the other 7 are summarized in Table 2.

Among the 7 static features, `has_modifier`, `has_call`, `has_balance`, `callee_external` and `can_send_eth` are static features. We collect them by utilizing static analysis techniques. The feature `has_modifier` is designed to identify existing program guards. In smart contract programs, the function modifier is often used to guard a function from arbitrary access. That is, a function with modifier is less like a vulnerable one. Therefore, we make the modifier as a counter-feature to avoid false alarms. Feature `has_call` and feature `has_balance` are designed to identify external calls and balance check operations. These two features are closely connected with transfer operations. We prepare them to better locate the transfer behavior and narrow search space. Feature `callee_external` provides important information on whether the function has external callees. This feature is used to capture risky calls. In smart contracts, cross-contract calls are prone to be exploited by attackers. Feature `can_send_eth` extracts static information (e.g., whether the function has transfer operation) to figure out whether the function has ability to send ethers to others. Considering the vulnerable functions often have risky transfer operations, this feature can help filtering out benign functions and reduce false positive reports.

The remaining three features, i.e., `has_delegate` and `has_tx_origin` correspond to particular key opcodes used in vulnerabilities. Specifically, feature `has_delegate` corresponds to the opcode `DELEGATECALL` in delegatecall vulnerabilities, feature `has_tx_origin` corresponds to the opcode `ORIGIN` in tx-origin vulnerabilities. These two features are specifically designed for the two vulnerabilities, as their names suggest. Note that the features can be easily updated to support detection on new vulnerabilities. If the new vulnerability shares similar mechanism with the above three vulnerabilities or is closely related to them, the existing features can be directly adopted; otherwise, one or two new specific features highly correlated with the new type of vulnerability should be added. The 7 static features are combined with word vectors, which together form the input to our ML models for further training.

4.3 Model Selection

In this section, we train and evaluate diverse candidate models, based on which we select the best one to guide fuzzers. To achieve this, one challenge we have to address first is the dataset imbalance. In particular, there are 1,162 vulnerabilities and 98,977 benign contracts. This is not rare in ML-based vulnerability detection tasks [42], [43]. In fact, our dataset endures imbalance in rate of 1:126 for reentrancy, 1:2,502 for delegatecall and 1:298 for tx-origin. Such imbalanced dataset can hardly be used for training.

To address the challenge, we first eliminate the duplicated data. In fact, we found 73,666 word vectors are exactly same to others. These samples are different in source code, but after they are compiled, extracted and transformed into vectors, they share the same values, because most of them are syntactically identical clones [44] at source code level. After our remedy, data imbalance comes to 1:31 for reentrancy, 1:189 for delegatecall and 1:141 for tx-origin. Still the dataset is highly imbalanced.

As studied in [45], the imbalance can be alleviated by data sampling strategies. However, we find that sampling strategies like oversampling [46] can hardly improve the precision and recall of models because the strategy introduces too much polluted data instead of real vulnerabilities.

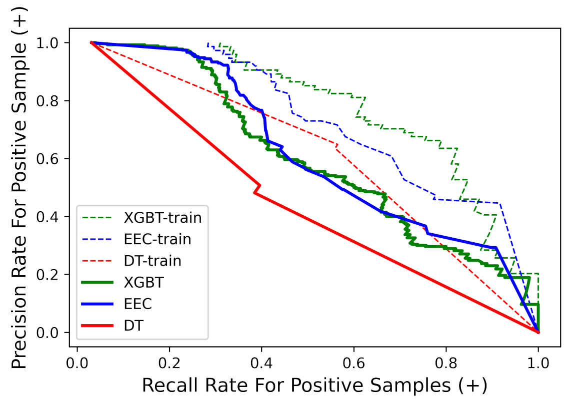


Fig. 6: The P-R Curve of models. The dashed lines represent performance on training set, while the solid lines represent performance on validation set.

TABLE 3: The performance of evaluated ML models.

Model Name	Precision	Recall
EasyEnsembleClassifier	26%	95%
XGBoost	66%	48%
DecisionTree	70%	43%
SupportVectorMachine	60%	14%
KNeighbors	50%	43%
NaiveBayes	50%	59%
LogiticRegression	53%	38%

We then attempt to evaluate models to select one that fits the imbalanced data well. Note that to counteract the impact of different ML models, we try to cover as many candidate ML methods as possible, among which we select the best one. The models we evaluated including tree-based models XGBT [38], EEC [39], Decision Tree (DT), and other representative ML models like Logistic Regression, Bayes Models, SVMs and LSTM [47]. The performance of the models can be found at Table 3. We find that the tree-based models achieve better precision and recall than others. Other non-tree-based models are biased towards the major class and hence show very poor classification rates on minor classes. Therefore, we select XGBT, EEC and DT as the candidate models.

The precision-recall curves of the three models on positive cases are shown on Figure 6. In this figure, the dashed lines denote models fitting with validation set and solid lines denote fitting with testing set. Intuitively, model XGBT and model EEC achieve better performance with similar P-R curves. However, EEC performs much better than XGBT in recall. In fact, model XGBT holds a precision rate of 66% and a recall rate of 48%. Comparatively, model EEC achieves a precision rate of 26% and a recall rate of 95%. We remark that our goal is not to train a model that is very accurate, but rather a model that allows us to filter as many benign contracts as possible without missing real vulnerabilities. Therefore, we select the EEC model for further guiding the fuzzing process.

4.4 Model Robustness Evaluation

To further evaluate the robustness of our selected model and to assess that to how much extent can our

TABLE 4: The coverage rate (CR) score of ML model on other tools.

	CR(Slither)	CR(Securify)	CR(Solhint)
Reentrancy	83.6%	81.1%	86.3%
Tx-origin	91.9%	N.A.	75.1%
Delegatecall	90.6%	N.A.	N.A.

model represent existing analyzers, we conduct evaluation of comparing the vulnerability detection on unknown dataset between our model and other state-of-the-art static analyzers. The evaluation dataset is download from a prominent third-party blockchain security team (https://github.com/tintinweb/smart-contract-sanctuary). We select smart contracts released in version 0.4.24 and 0.4.25 (i.e., the majority versions of existing smart contract applications [48]) and remove the contracts which has been used in our previous model training and model selection. After all, we get 78,499 contracts in total for evaluation.

Definition 7 (Coverage Rate of ML Model on Another Tool). Given the true positive reports of ML model R_m, the true positive reports of another tool R_t, a coverage rate of ML model $CR(t)$ on the tool is calculated as:

$$CR(t) = (R_m \cap R_t)/R_t \tag{1}$$

The results are listed in Table 4. Here, we use the coverage rate (CR) to evaluate the representativeness of our model regarding the three vulnerabilities. Specifically, the coverage rate measures how much reports of ML model are intersected with static analyzer tools. The coverage rate CR is calculate as listed in Definition 7. The N.A. in the table denotes that the detection of this vulnerability is not support by the analyzer.

Our evaluation results show that the reports of our tool can cover a majority of reports of other tools. Specifically, the trained ML model can well approximate the capability of each static tool used in vulnerability labeling and model training. For example, 81.1% of true positive reports of SECURIFY on reentrancy are also contained in our ML model's reports. Besides, 75.1% of true positive reports of SOLHINT on Tx-origin and 90.6% of true positive reports of SLITHER on Delegatecall are also covered.

Algorithm 1: Machine learning guided fuzzing

```
input  : IS, all the input smart contract source code
input  : M, suspicious function detection ML model
input  : TRs ← ∅, the set of potentially vulnerable
         function execution paths
output : V ← ∅, the set of vulnerable paths
1  F_s ← IS.getFunctionList()
2  // get the functions in a contract
3  foreach function f ∈ F_s do
4      if ifIsSuspiciousFunction(f, M) is True then
5          // employ ML models to predict
           whether the function is suspicious
6          S_func ← getFuncPriorityScore(f)
7          S_caller ← getCallerPriorityScore(f)
8          TRs ← TRs ∪ {f, S_func, S_caller}
9          // get scores for each function
10 PTR ← PrioritizationAlgorithm(TRs)
11 // Prioritized paths
12 V ← ∅
13 // the output vulnerability list
14 while not timeout do
15     T ← PTR.pop()
16     // pop up trace with higher priority
17     FuzzingResult ← Fuzzing(T)
18     if FuzzingResult is Vulnerable then
19         V ← V ∪ {T}
20     else
21         continue
22 return V
```

```
contract Wallet{
  function withdraw(address addr, uint value){
    addr.transfer(value);
  }
  function changeOwner(address[] addrArray,
      uint idx) public{
    require(msg.sender == owner);
    owner = addrArray[idx];
    withdraw(owner, this.balance);
  } }
contract Logic{
  function logTrans(address addr_w, address
      _exec, uint _value, bytes infor) public{
    Wallet(addr_w).withdraw(_exec, _value);
  } }
```

Fig. 7: An example of prioritizing paths.

5 GUIDED CROSS-CONTRACT FUZZING

5.1 Guidance Algorithm

The pretrained models are applied to guide fuzzers in the ways that the predictions are utilized to ❶ locate suspicious functions and ❷ combine with static information for path prioritization.

Our guidance is based on both model predictions and the priority scores computed from static features. The reason is that even with the machine learning model filtering, the search space is still rather large, which is evidenced by the large number of paths explored by SFUZZ (e.g., the 2,596 suspicious functions have 873 possibly vulnerable paths), and thus we propose to first prioritize the path.

The overall process of our guided fuzzing can be found at Algorithm 1. In this algorithm, we first retrieve function list of an input source at line 1. Next, from line 3 to line 8, we calculate the path priority based on two scores (i.e., function priority scores and caller priority scores) for each path. Both scores are designed for prioritizing suspicious functions. After the calculation, the results are saved together with the function itself. In line 10, we prioritize the suspicious function paths. The prioritization algorithm can be found at Algorithm 2. The trace with higher priority will be first tested by fuzzers. Finally, from line 14 to line 21, we pop up a candidate trace from prioritized list and employ fuzzers to conduct focus fuzzing. The fuzzing process will not end until it reaches an timeout limitation. The found vulnerability will be return as final result.

The details of our prioritization algorithm are shown in Algorithm 2. The input of the algorithm is the functions and their corresponding priority scores. The scores are calculated in Algorithm 1. The output of the algorithm is the prioritized vulnerable paths. Specifically, the first step of the algorithm is

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3182373

Algorithm 2: Priorization Algorithm

input : M, The trained machine learning model
input : TRs, functions and their priority scores
output: PTR, the set of prioritized vulnerable paths

1 **while** $isNotEmpty(TRs)$ **do**
2     $TRs \leftarrow sortByFunctionPriority(TRs)$
3   function $f \leftarrow TRs.pop()$
4   paths $Ps \leftarrow getAllPaths(f)$
5   **while** $isNotEmpty(Ps)$ **do**
6         $Ps \leftarrow sortByCallerPriority(Ps)$
7         $P \leftarrow Ps.pop()$
8         $PTR \leftarrow PTR \cup P$
9 **return** PTR

getting the prioritized function based on the function priority score, as shown in line 2 and line 3. The functions with lower function priority scores will be prioritized. Next, we sort all call paths (no matter cross-contract or non-cross-contract call) which are correlated to the function, as shown from line 4 to line 6. We pop up the call path which has the highest priority and add it to the prioritized path set. The prioritized path set will guide fuzzer to test call path in a certain order.

To summarize, the goal of our guidance algorithm is to prioritize cross-contract paths, which are penetrable but usually overlooked by previous practice [15], [14], and to further improve the fuzzing testing efficiency on cross-contract vulnerabilities.

5.2 Priority Score

Generally, the path priority consists of two parts: *function priority* and *caller priority*. The function priority is for evaluating the complexity of function and the caller priority is designed to measure the cost to traverse a path.

Function Priority. We collect static features of functions to compute function priority. After that, a priority score can be obtained. The lower score denotes higher priority.

We first mark the suspicious functions by model predictions. A suspicious function is likely to contain vulnerabilities so it is provided with higher priority. We implement this as a factor f_s which equals 0.5 for suspicious function otherwise 1 for benign functions. For example, in Figure 7, the function `withdraw` is predicted as suspicious so that the factor f_s equals 0.5.

Next, we compute the caller dimensionality S_C. The dimensionality is the number of callers of a function. In cross-contract fuzzing, a function with multiple callers requires more testing time to traverse all paths. For example, in Figure 7, function `withdraw` in contract `Wallet` has an internal caller `changeOwner` and an external caller `logTrans`, thus the dimensionality of this function is 2.

The parameter dimensionality S_P is set to measure the complexity of parameters. The functions with complex parameters (i.e., array, bytes and address parameters) are assigned with lower priority, because these parameters often increase the difficulty of penetrating a function. Specifically, one parameter has 1 dimensionality except for the complex parameters, i.e., they have 2 dimensionalities. The parameter dimensionality of a function is the sum of parameters dimensionalities. For example, in Figure 7, function `withdraw` and `changeOwner` both have an address and an integer parameter thus their dimensionality is 3. Function `logTrans` has two addresses, a byte and an integer parameter, so the dimensionality is 7.

Definition 8 (Function Priority Score). Given the suspicious factor f_s, the caller dimensionality score S_C and the parameter dimensionality score S_P, a function priority score S_{func} is calculated as:

$$S_{func} = f_s \times (S_C + 1) \times (S_P + 1) \tag{2}$$

In this formula, we add 1 to the caller dimensionality and parameter dimensionality to avoid the overall score to be 0. The priority scores in Figure 7 are: function `withdraw` = 6, function `changeOwner` = 4, function `logTrans` = 8. The results show that function `changeOwner` has highest priority because function `withdraw` has two callers to traverse meanwhile function `logTrans` is more difficult for penetration than `changeOwner`.

Caller Priority. We traverse every caller of a function and collect their static features, based on which we compute the priority score to decide which caller to test first. Firstly, the number of branch statements (e.g., `if`, `for` and `while`) and assertions (e.g., `require` and `assert`) are counted to measure condition complexity $Comp$ to describe the difficulties to bypass the conditions. The path with more conditions is in lower priority. For example, in Figure 7, function `withdraw` has two callers. One caller `changeOwner` has an assertion at line 6, so the complexity is 1. The other caller `logTrans` contains no conditions thus the complexity is 0.

Next, we count the condition distance. sFuzz selects seed according to branch-distance only, which is not ideal for identifying the three particular kinds of cross-contract vulnerabilities that we focus on in this work. Thus, we propose to consider not only branch distance but also this condition distance $CondDis$. This distance is intuitively the number of statements from entry to condition. In case of the function has more than one conditions, the distance is the number of statements between entry and first condition. For example, in Figure 7, the condition distance of `changeOwner` is 1 and the condition distance of `logTrans` is 0.

Definition 9 (Caller Priority Score). Given the condition distance $CondDis$ and the path condition complexity $Comp$, a path priority score S_{caller} is calculated as:

$$S_{caller} = (CondDis + 1) \times (Comp + 1) \tag{3}$$

Finally, the caller priority score is computed based on condition complexity and condition distance, as shown in Definition 9. The complexity and distance add 1 so that the overall score is not 0. The caller priority scores in Figure 7 are: `logTrans` $\rightarrow$ `withdraw` = 1, `changeOwner` $\rightarrow$ `withdraw` = 4. Function `changeOwner` has identity check at line 6, which increase the difficulty to penetrate. Thus, the other path from `logTrans` to `withdraw` is prior.

5.3 Cross-contract Fuzzing

Given the prioritized paths, we utilized cross-contract fuzzing to improve fuzzing efficiency. Here, we implement this fuzzing technique by the following steps: 1) The contracts

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3182373

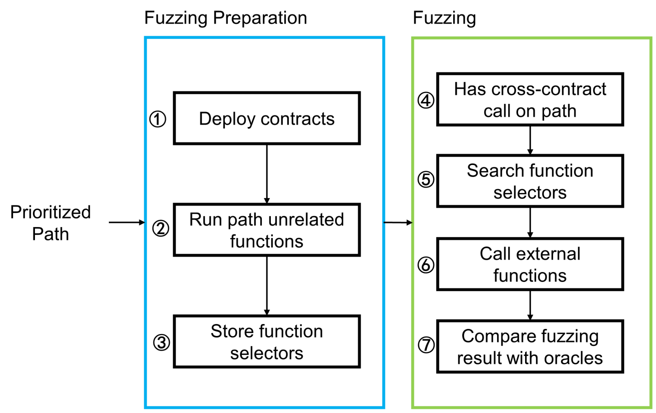


Fig. 8: The cross-contract fuzzing process.

under test should be deployed on EVM. As shown in Figure 8, the fuzzer will first deploy all contracts on a local private chain to facilitate cross-contract calls among contracts. 2) The path-unrelated functions will be called. Here, the path-unrelated functions denote functions that do not appear in the input prioritized paths. We run them first to initialize state variables of a contract. 3) We store the function selectors appeared in all contracts. The function selector is the unique identity recognizer of a function. It is usually encoded in 4-byte hex code [49]. 4) The fuzzer checks whether there is a cross-contract call. If not, the following step 5 and step 6 will be skipped. 5) The fuzzer automatically searches local states to find out correct function selectors, and then directly trigger a cross-contract call to the target function in step 6. 7) The fuzzer compares the execution results against the detection rules and output reports.

6 Evaluation

xFuzz is implemented in Python and C with 3298 lines of code. All experiments are run on a computer which is running Ubuntu 18.04 LTS and equipped with Intel Xeon E5-2620v4, 32GB memories and 2TB HDD.

For the baseline comparison, xFuzz is compared with the state-of-art fuzzer sFuzz [14], a previously published testing engine ContractFuzzer [15] and a static cross-contract analysis tool Clairvoyance [19]. The recently published tool Echidna [16] relies on manually written testing oracles, which may lead to different testing results depending on developer's expertise. Thus, it is not compared. Other tools (like Harvey [21]) are not publicly available for evaluation, and thus are not included in our evaluations. We systematically run all four tools on the contract datasets. Notably, to verify the authenticity of the vulnerability reports, we invite senior technical experts from security department of our industry partner to check vulnerable code. Our evaluation aims at investigating the following research questions (RQs).

RQ1. How effective is xFuzz in detecting cross-contract vulnerabilities?

RQ2. To what extent the machine learning models and the path prioritization contribute to reducing the search space?

RQ3. What are the overhead of xFuzz, compared to the vanilla sFuzz?

TABLE 5: Evaluations on Dataset1. The ✔ represents the tool successfully finds vulnerability in this function, otherwise the tool is marked with ✖.

Address	ContractFuzzer	xFuzz	sFuzz
0x7a8721a9	✖	✔	✖
0x4e73b32e	✖	✔	✔
0xb5e1b1ee	✖	✔	✔
0xaae1f51c	✖	✔	✔
0x7541b76c	✖	✔	✖
...	...	...	...
Summary	ContractFuzzer	xFuzz	sFuzz
	0/18	9/18	5/18

RQ4. Can xFuzz discover real-world unknown cross-contract vulnerabilities, and what are the reasons for false negatives?

6.1 Dataset Preparation

Our evaluation dataset includes smart contracts from three sources: 1) datasets from previously published works (e.g., [30] and [31]); 2) smart contract vulnerability websites with good reputation (e.g., [40]); 3) smart contracts downloaded from Etherscan. The dataset is carefully checked to remove duplicate contracts with dataset used in our machine learning training. Specifically, the **DataSet1** includes contracts from previous works and famous websites. After we remove duplicate contracts and toy-contract (i.e., those which are not deployed on real world chains), we collect 18 labeled reentrancy vulnerabilities. To enrich the evaluation dataset, our **Dataset2** includes contracts downloaded from Etherscan. We remove contracts without external calls (they are irrelevant to cross-contract vulnerabilities) and contracts that are not developed by using Solidity 0.4.24 and 0.4.25 (i.e., the most two popular versions of Solidity [48]). In the end, 7,391 contracts are collected in **Dataset2**. The source code of the above datasets are publicly available in our website [32] so that the evaluations are reproducible, benefiting further research.

6.2 RQ1: Vulnerability Detection Effectiveness

We first conduct evaluations on **Dataset1** by comparing three tools ContractFuzzer, sFuzz and xFuzz. The Clairvoyance is not included because it is a static analysis tool. For the sake of page space, we present a part of the results in Table 5 with an overall summary and leave the whole list available at here[1].

In this evaluation, ContractFuzzer fail to find a vulnerability among the contracts. sFuzz missed 3 vulnerabilities and outputted 9 incorrect reports. Comparatively, xFuzz missed 2 vulnerabilities and outputted 6 incorrect reports. The reason of the missed vulnerabilities and incorrect reports lies on the difficult branch conditions (e.g., an `if` statement with 3 conditions) which blocks the fuzzer to traverse vulnerable branches. Note that xFuzz is equipped with model guidance so that it can focus on fuzzing suspicious functions and find more vulnerabilities than sFuzz.

1. https://anonymous.4open.science/r/xFuzzforReview-ICSE/Evaluation\%20on\%20Open-dataset.pdf

TABLE 6: Performance of xFuzz, Clairvoyance (C.V.), ContractFuzzer (C.F.), sFuzz on cross-contract vulnerabilities.

	reentrancy			delegatecall			tx-origin		
	P%	R%	#N	P%	R%	#N	P%	R%	#N
C.F.	0	0	0	0	0	0	0	0	0
sFuzz	0	0	0	0	0	0	0	0	0
C.V.	43.7	43.7	16	0	0	0	0	0	0
xFuzz	100	81.2	13	100	100	3	100	100	2

While we compare our tool with existing works on publicly available **Dataset1**, the dataset only provides non-cross-contract labels thus cannot be used to verify our detection ability on cross-contract ones. To complete this, we further evaluate the effectiveness of cross-contract and non-cross-contract fuzzing on **Dataset2**. To reduce the effect of randomness, we repeat each setting 20 times, and report the averaged results.

6.2.1 *Cross-contract Vulnerability.*

The results are summarized in Table 6. Note that the "P%" and "R%" represent precision rate and recall rate, "#N" is the number of vulnerability reports. "C.V." means Clairvoyance and "C.F." means ContractFuzzer. Cross-contract vulnerabilities are currently not supported by ContractFuzzer, sFuzz and thus they report no vulnerabilities detected.

Precision. Clairvoyance managed to find 7 true cross-contract reentrancy vulnerabilities. In comparison, xFuzz found 9 cross-contract reentrancy, 3 cross-contract delegatecall and 2 cross-contract tx-origin vulnerabilities. The two tools found 21 cross-contract vulnerabilities in total. Clairvoyance report 16 vulnerabilities but only 43.7% of them are true positives. In contrast, xFuzz generates 18 (13+3+2) reports of three types of cross-contract vulnerabilities and all of them are true positives. The reason of the high false positive rate of Clairvoyance is mainly due to its static analysis based approach, without runtime validation. We further check the 18 vulnerabilities on some third-party security expose websites [50], [40], [31] and we find 15 of them are not flagged.

Recall. The 9 vulnerabilities missed by Clairvoyance are all resulted from the abuse of detection rules, i.e., the vulnerable contracts are filtered out by unsound rules. In total, 3 cross-contract vulnerabilities are missed by xFuzz. A close investigation shows that they are missed due to the complex path conditions, which blocks the input from penetrating the function. We also carefully check false negatives missed by xFuzz, and find they are not reported by ConractFuzzer and sFuzz as well. While existing works all fail to penetrate the complex path conditions, we believe this limitation can be addressed by future works.

6.2.2 *Non-Cross-contract Vulnerability.*

The experiment results show that xFuzz improves detection of non-cross-contract vulnerabilities as well (see Table 7). For reentrancy, ContractFuzzer achieves the best 100% precision rate but the worst 1.7% recall rate. sFuzz and Clairvoyance identified 33.5% and 40.4% vulnerabilities. xFuzz has a precision rate of 95.5%, which is slightly lower than that of ContractFuzzer, and more importantly, the bests recall rate of 84.2%. xFuzz exhibits strong capability in detecting vulnerabilities by finding a total of 209 (149+35+25) vulnerabilities.

TABLE 7: Performance of xFuzz, Clairvoyance, ContractFuzzer and sFuzz on non-cross-contract evaluations.

	reentrancy			delegatecall			tx-origin		
	P%	R%	#N	P%	R%	#N	P%	R%	#N
C.F.	100	1.7	3	0	0	0	0	0	0
sFuzz	84.2	33.5	70	100	54.3	19	0	0	0
C.V.	48.3	40.4	145	0	0	0	0	0	0
xFuzz	95.5	84.6	156	100	100	35	100	100	25

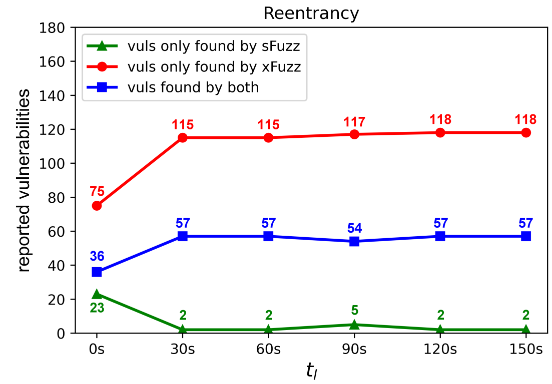


Fig. 9: Comparison of reported vulnerabilities between xFuzz and sFuzz regarding reentrancy.

Precision. For reentrancy, Clairvoyance reports 75 false positives, because of ❶ the abuse of detection rules and ❷ unexpected jump to unreachable paths due to program errors. The 11 false positives of sFuzz are due to the misconceived ether transfer. sFuzz captures ether transfers to locate dangerous calls. However, the ethers from attacker to victim is also falsely captured. The 7 false alarms of xFuzz are due to the mistakes of contract programmers by calling a nonexistent functions. These calls are however misconceived as vulnerabilities by xFuzz.

Recall. Clairvoyance missed 59.6% of the true positives. The root cause is the adoption of unsound rules during static analysis. sFuzz missed 117 reentrancy vulnerabilities and 16 delegatecall vulnerabilities due to *(1)* timeout and *(2)* incapability to find feasible paths to the vulnerability. xFuzz missed 27 vulnerabilities due to complex path conditions.

> **Answer to RQ1:** Our tool xFuzz achieves a precision of 95.5% and a recall of 84.6%. Among the evaluated four methods, xFuzz achieves the best recall. Besides, xFuzz successfully finds 209 real-world non-cross-contract vulnerabilities as well as 18 real-world cross-contract vulnerabilities.

6.3 RQ2: The Effectiveness of Guided Testing

This RQ investigates the usefulness of the ML model and path prioritization for the guidance of fuzzing. To answer this RQ, we compare sFuzz with a customized version of xFuzz, i.e., which differs from sFuzz only by adopting the ML

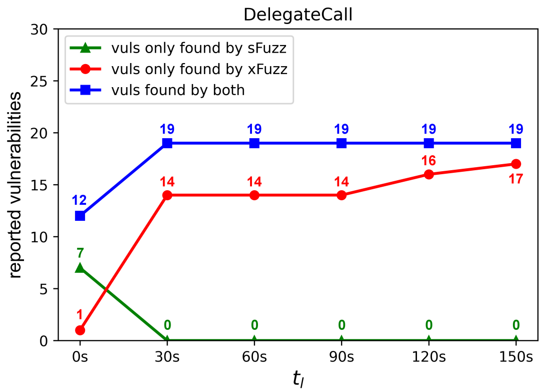


Fig. 10: Comparison of reported vulnerabilities between xFuzz and sFuzz regarding delegatecall.

model (without focusing on cross-contract vulnerabilities). The intuition is to check whether the ML model enables us to reduce the time spent on benign contracts and thus reveal vulnerabilities more efficiently. That is, we implement xFuzz such that each contract is only allowed to be fuzzed for t_l seconds if the ML model considers the contract benign; or otherwise, 180 seconds, which is also the time limit adopted in sFuzz. Note that if t_l is 0, the contract is skipped entirely when it is predicted to be benign by the ML model. The goal is to see whether we can set t_l to be a value smaller than 180 safely (i.e., without missing vulnerabilities). We thus systematically vary the value of t_l and observe the number of identified vulnerabilities.

The results are summarized in Figure 9 and Figure 10. Note that the tx-origin vulnerability is not included since it is not supported by sFuzz. The red line represents vulnerabilities only found by xFuzz, the green line represents vulnerabilities only reported by sFuzz and the blue line denotes the reports shared by both two tools. We can see that the curves climb/drop sharply at the beginning and then saturate/flatten after 30s, indicating that most vulnerabilities are found in the first 30s.

We observe that when t_l is set to 0s (i.e., contracts predicted as benign are skipped entirely), xFuzz still detects 82.8% (i.e., 111 out of 134, or equivalently 166% of that of sFuzz) of the reentrancy vulnerabilities as well as 65.0% of the delegatecall vulnerability (13 out of 20). The result further improves if we set t_l to be 30 seconds, i.e., almost all (except 2 out of 174 reentrancy vulnerabilities; and none of the delegatecall vulnerabilities) are identified. Based on the result, we conclude that the ML model indeed enables to reduce fuzzing time on likely benign contracts significantly (i.e., from 180 seconds to 30 seconds) without missing almost any vulnerability.

The Effectiveness of Path Prioritization. To evaluate the relevance of path prioritization, we further analyze the results of the customized version of xFuzz as discussed above. Recall that path prioritization allows us to explore likely vulnerable paths before the remaining. Thus, if path prioritization works, we would expect that the vulnerabilities are mostly found in paths, where xFuzz explores first. We thus systematically count the number of vulnerabilities found in the first 10 paths which are explored by xFuzz. The results

TABLE 8: The paths reported by xFuzz and sFuzz. The vulnerable paths found by the two tools are counted respectively.

Found by	Vul	Total	Number in the Top	
			Top10	Other
xFuzz	Reentrancy	172	152	20
sFuzz	Reentrancy	59	57	2
xFuzz	Delegatecall	33	32	1
sFuzz	Delegatecall	19	19	0

TABLE 9: The time cost of each step in fuzzing procedures.

		sFuzz	C.V.	xFuzz
MPT(min)	Reentrancy	N.A.	N.A.	630.6
	Delegatecall	N.A.	N.A.	630.6
	Tx-origin	N.A.	N.A.	630.6
ST(min)	Reentrancy	21,930.0	N.A.	3,621.0
	Delegatecall	22,131.0	N.A.	3,678.0
	Tx-origin	N.A.	N.A.	3,683.0
DT(min)	Reentrancy	54.1	246.2	86.6
	Delegatecall	2.8	N.A.	4.2
	Tx-origin	N.A.	N.A.	2.9
Total(min)	Reentrancy	21,984.1	246.2	4,338.2
	Delegatecall	22,133.8	N.A.	4,312.8
	Tx-origin	N.A.	N.A.	4,316.5

are summarized in Table 8, where column "Top 10" shows the number of vulnerabilities detected in the first 10 paths explored.

The results show that, xFuzz finds a total of 152 (out of 172) reentrancy vulnerabilities in the first 10 explored paths. In particular, the number of found vulnerabilities in the first 10 explored paths by xFuzz is almost three times as many as that by sFuzz. Similarly, xFuzz also finds 32 (out of 33) delegatecall vulnerabilities in the first 10 explored paths. The results thus clearly suggest that path prioritization allows us to focus on relevant paths effectively, which has practical consequence on fuzzing large contracts.

Answer to RQ2: The ML model enables us to significantly reduce the fuzzing time on likely benign contracts without missing almost any vulnerabilities. Furthermore, most vulnerabilities are detected efficiently through our path prioritization. Overall, xFuzz finds *twice* as many reentrancy or delegatecall vulnerabilities as sFuzz.

6.4 RQ3: Detection Efficiency

Next, we evaluate the efficiency of our approach. We record time taken for each step during fuzzing and the results are summarized in Table 9. To eliminate randomness during fuzzing, we replay our experiments for five times and report the averaged results. In this table, "MPT" means model prediction time; "ST" means search time for vulnerable paths during fuzzing; "DT" means detection time for Clairvoyance and fuzzing time for the fuzzers. "N.A." means that the tool has no such step in fuzzing or the vulnerability is currently not supported by it, and thus the time is not recorded.

The efficiency of our method (i.e., by reducing the search space) is evidenced as the results show that xFuzz is

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3182373


```
function buyOne(address _exchange, uint256
    _value, bytes _data) payable public
{
  ...
  buyInternal(_exchange, _value, _data);
}
function buyInternal(address _exc, uint256
    _value, bytes _data) internal
{
  ...
  require(_exc.call.value(_value)(_data));
  balances[msg.sender] = balances[msg.sender
      ].sub(_value);
}
```

Fig. 11: A real-world reentrancy vulnerability found by XFUZZ, in which the vulnerable path relies on internal calls.

obviously faster than SFUZZ, i.e., saving 80% of the time. The main reason for the saving is due to the saving on the search time (i.e., 80% reduction). We also observe that XFUZZ is slightly slower than SFUZZ in terms of the effective fuzzing time, i.e., an additional 32.5 (86.6-54.1) min is used for fuzzing cross-contract vulnerabilities. This is expected as the number of paths is much more (even after the reduction thanks to the ML model and path prioritization) than that in the presence of more than 2 interacting contracts. Note that CLAIRVOYANCE is faster than all tools because this tool is a static detector without perform runtime execution of contracts.

> **Answer to RQ3:** Owing to the reduced search space of suspicious functions, the guided fuzzer XFUZZ saves over 80% of searching time and reports more vulnerabilities than SFUZZ with less than 20% of the time.

6.5 RQ4: Real-world Case Studies

In this section, we present 2 typical vulnerabilities reported by XFUZZ to qualitatively show why XFUZZ works. In general, the ML model and path prioritization help XFUZZ find vulnerabilities in three ways, i.e., ❶ locate vulnerable functions, ❷ identify paths from internal calls and ❸ identify feasible paths from external calls.

Real-world Case 1: XFUZZ is enhanced with path prioritization, which enables it to focus on vulnerabilities related to internal calls. In Figure 11[2], the modifier `internal` limits the access only to internal member functions. The attacker can however steal ethers by path `buyOne` → `buyInternal`. By applying XFUZZ, the vulnerability is identified in 0.05 seconds and the vulnerable path is also efficiently exposed.

Real-world Case 2: The path prioritization also enables XFUZZ to find cross-contract vulnerabilities efficiently. For example, a real-world cross-contract vulnerability[3] is shown in Figure 12. This example is for auditing transactions in real-world and involves with over 2,000 dollars. In this example, function `registerAudit` has a cross-contract call to a public address `CSolidStamp` at line 13, which intends to forward the call to function `audContract`. While this function is only allowed to be accessed by the registered functions, as limited by modifier `onlyRegister`, we can bypass this restriction

2. deployed at 0x0695B9EA62C647E7621C84D12EFC9F2E0CDF5F72
3. deployed at 0x165CFB9CCF8B185E03205AB4118EA6AFBDBA9203

```
1  contract SolidStamp{
2    function audContract(address _auditor) public
         onlyRegister
3    {
4      ...
5      _auditor.transfer(reward.sub(commissionKept
           ));
6    }
7  }
8  contract SolidStampRegister{
9    address public CSolidStamp;
10   function registerAudit(bytes32 _codeHash)
         public
11   {
12     ...
13     SolidStamp(CSolidStamp).audContract(msg.
           sender);
14   }
15 }
```

Fig. 12: A cross-contract vulnerability found by XFUZZ. This contract is used in auditing transactions in real-world.

```
if ((random()%2==1) && (msg.value == 1 ether)
    && (!locked))
\\at 0x11F4306f9812B80E75C1411C1cf296b04917b2f0

require(msg.value == 0 || (_amount == msg.value
    && etherTokens[fromToken]));
\\at 0x1a5f170802824e44181b6727e5447950880187ab
```

Fig. 13: Complex path conditions involving with multiple variables and values.

by a cross-contract call `registerAudit` → `audContrat`. Eventually, an attacker would be able to steals the ethers in seconds.

Real-world Case 4:During our investigation on the experiment results, we gain the insights that XFUZZ can be further improved in terms of handling complex path conditions. Complex path conditions often lead to prolonged fuzzing time or blocking penetration altogether. We identified a total of 3 cross-contract and 24 non-cross-contract vulnerabilities that are missed due to such a reason. Two of such complex condition examples (from two real-word false negatives of XFUZZ) are shown in Figure 13. Function calls, values, variables and arrays are involved in the conditions. These conditions are difficult to satisfy for XFUZZ and fuzzers in general (e.g., SFUZZ failed to penetrate these paths too). This problem can be potentially addressed by integrating XFUZZ with a theorem prover such that Z3 [51] which is tasked to solve these path conditions. That is, a hybrid fuzzing approach that integrates symbolic execution in a lightweight manner is likely to further improve XFUZZ.

> **Answer to RQ4:** With the help of model predictions and path prioritization, XFUZZ is capable of rapidly locating vulnerabilities in real-world contracts. The main reason for false negatives is complex path conditions, which could be potentially addressed through integrating hybrid fuzzing into XFUZZ.

7 RELATED WORK

In this section, we discuss works that are most relevant to ours.

Program analysis. We draw valuable development experience and domain specific knowledge from existing work [8], [10], [3], [4], [5]. Among them, SLITHER [10], OYENTE [8] and Atzei *et al.* [5] provide a transparent overlook on smart contracts detection and enhance our understanding on vulnerabilities. Chen *et al.* [3] and Durieux *et al.* [4] offer evaluations on the state-of-the-arts, which helps us find the limitation of existing tools.

Cross-contract vulnerability. Our study is closely related to previous works focusing on interactions between multiple contracts. Zhou *et al.* [52] present work to analyze relevance between smart contract files, which inspires us to focus on cross-contract interactions. He *et al.* [24] report that existing tools fail to exercise functions that can only execute at deeper states. Xue *et al.* [19] studied cross-contract reentrancy vulnerability. They propose to construct ICFG (combining CFGs with call graphs) then track vulnerability by taint analysis.

Smart contract testing. Our study is also relevant to previous fuzzing work on smart contracts. Smart contract testing plays an important role in smart contract security. Zou *et al.* [7] report that over 85% of developers intend to do heavy testing when programming. The work of Jiang *et al.* [15] makes the early attempt to fuzz smart contracts. CONTRACTFUZZER instruments Ethereum virtual machine and then collects execution logs for further analysis. Wüstholz *et al.* present guided fuzzer to better mutate inputs. Similar method is implemented by He *et al.* [24]. They propose to learn fuzzing strategies from the inputs generated from a symbolic expert. The above two methods inspire us to leverage a guider to reduce search space. Tai D *et al.* [14] implement a user-friendly AFL fuzzing tool for smart contracts, based on which we build our fuzzing framework. Different from these existing work, our work makes a special focus on proposing novel ML-guided method for fuzzing cross-contract vulnerabilities, which is highly important but largely untouched by existing work. Additionally, our comprehensive evaluation demonstrates that our proposed technique indeed outperforms the state-of-the-arts in detecting cross-contract vulnerabilities.

Machine learning practice. This work is also inspired by previous work [53], [54], [55]. In their work, they propose learning behavior automata to facilitate vulnerability detection. Zhuang *et al.* [56] propose to build graph networks on smart contracts to extend understanding of malicious attacks. Their work inspires us to introduce machine learning method for detection. We also improve our model selection by inspiration of work of Liu *et al.* [39]. Their algorithm helps us select best models with satisfactory performance on recall and precision on highly imbalanced dataset. Yan *et al.* [55] have proposed a method to mimic the cognitive process of human experts. Their work inspires us to find the consensus of vulnerability evaluators to better train the machine learning models.

Smart contract security to society. Smart contract has drawn a number of security concerns since it came into being. As figured out by Zou *et al.* [7], over 75% of developers agree that the smart contract software has a much high security requirement than traditional software. According to [7], the reasons behind such requirement are: 1) The frequent operations on sensitive information (e.g., digital currencies, tokens); 2) The transactions are irreversible; 3) The deployed code cannot be modified. Considering the close connection between smart contract and financial activities, the security of smart contract security largely effects the stability of the society.

8 CONCLUSION

In this paper, we propose XFUZZ, a novel machine learning guided fuzzing framework for smart contracts, with a special focus on cross-contract vulnerabilities. We address two key challenges during its development: the search space of fuzzing is reduced, and cross-contract fuzzing is completed. The experiments demonstrate that XFUZZ is much faster and more effective than existing fuzzers and detectors. In future, we will extend our framework with more static approach to support more vulnerabilities.

REFERENCES

[1] V. K. DAS, "Top blockchain platforms of 2020," https://www.blockchain-council.org/blockchain/topblockchainplatformsof2020that\ \everyblockchainenthusiastmustknow/, 2020, online; accessed September 2020.

[2] Ethereum, "Ethereum daily transaction chart," https://etherscan.io/chart/tx, 2017, online; accessed 29 January 2017.

[3] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, 2020.

[4] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd ICSE*, 2020, pp. 530–541.

[5] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[6] O. G. Güçlütürk, "The dao hack explained: Unfortunate take-off of smart contracts," https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562, 2018, online; accessed 22 January 2018.

[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*, vol. 47, no. 10, pp. 2084–2106, 2019.

[8] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC CCS*, 2016, pp. 254–269.

[9] P. Tsankov, A. Dan, D. Drachsler-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.

[10] 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)*, 2019, pp. 8–15.

[11] Protofire, "Solhint," https://github.com/protofire/solhint, 2018, online; accessed September 2018.

[12] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *NDSS*, 2018.

[13] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *WETSEB*, 2018, pp. 9–16.

[14] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "Sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, New York, NY, USA, 2020, p. 778–788.

[15] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 259–269.

[16] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.

[17] Q. Zhang, Y. Wang, J. Li, and S. Ma, "Ethploit: From fuzzing to efficient exploit generation against smart contracts," in *2020 IEEE 27th SANER*. IEEE, 2020, pp. 116–126.

[18] J. Gao, H. Liu, Y. Li, C. Liu, Z. Yang, Q. Li, Z. Guan, and Z. Chen, "Towards automated testing of blockchain-based decentralized applications," in *IEEE/ACM 27th ICPC*, 2019, pp. 294–299.

[19] X. Yinxing, M. Mingliang, L. Yun, S. Yulei, Y. Jiaming, and P. Tianyong, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *2020 35rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[20] G. A. Oliva, A. E. Hassan, and Z. M. J. Jiang, "An exploratory study of smart contracts in the ethereum blockchain platform," *Empirical Software Engineering*, pp. 1–41, 2020.

[21] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.

[22] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: Identifying vulnerable code for vulnerability assessment through program metrics," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 60–71.

[23] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.

[24] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[25] S. T. Help, "7 principles of software testing: Defect clustering and pareto principle," https://www.softwaretestinghelp.com/7-principles-of-software-testing/, accessed March, 2021.

[26] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 415–427.

[27] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: what is the best choice?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 566–579.

[28] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[29] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[30] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: What is the best choice?" 2021.

[31] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," *arXiv preprint arXiv:2007.04771*, 2020.

[32] xFuzz, "Machine learning guided cross-contract fuzzing," https://anonymous.4open.science/r/xFuzzforReview-ICSE, 2020, online; accessed September 2020.

[33] ethervm, "Ethereum virtual machine opcodes," https://ethervm.io/, 2019, online; accessed September 2019.

[34] T. D. Nguyen, L. H. Pham, and J. Sun, "sguard: Towards fixing vulnerable smart contracts automatically," *arXiv preprint arXiv:2101.01917*, 2021.

[35] Protofire, "Decentralized application security project," https://dasp.co/, accessed September, 2018.

[36] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016.

[37] W. Drewry and T. Ormandy, "Flayer: Exposing application internals," 2007.

[38] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[39] X.-Y. Liu, J. Wu, and Z.-H. Zhou, "Exploratory undersampling for class-imbalance learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, pp. 539–550, 2008.

[40] S. C. Security, "Smart contract weakness classification registry," https://github.com/SmartContractSecurity/SWC-registry, 2019, online; accessed September 2019.

[41] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, 2019.

[42] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, pp. 103 184–103 197, 2019.

[43] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, 2016, p. 85–96.

[44] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, pp. 654–670, 2002.

[45] J. L. Leevy, T. M. Khoshgoftaar, R. A. Bauder, and N. Seliya, "A survey on addressing high-class imbalance in big data," *Journal of Big Data*, p. 42, 2018.

[46] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[47] Z. Huang, W. Xu, and K. Yu, "Bidirectional lstm-crf models for sequence tagging," *arXiv preprint arXiv:1508.01991*, 2015.

[48] Z. Tian, J. Tian, Z. Wang, Y. Chen, H. Xia, and L. Chen, "Landscape estimation of solidity version usage on ethereum via version identification," *International Journal of Intelligent Systems*, vol. 37, no. 1, pp. 450–477, 2022.

[49] S. Contract, "Function selector," https://solidity-by-example.org/function-selector/, accessed March, 2021.

[50] Dedaub, "Security technology for smart contracts," https://contract-library.com/, 2020, online; accessed 29 January 2020.

[51] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[52] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.

[53] H. Xiao, J. Sun, Y. Liu, S.-W. Lin, and C. Sun, "Tzuyu: Learning stateful typestates," in *2013 28th IEEE/ACM ASE*. IEEE, 2013, pp. 432–442.

[54] Y. Xue, J. Wang, Y. Liu, H. Xiao, J. Sun, and M. Chandramohan, "Detection and classification of malicious javascript via attack behavior modelling," in *Proceedings of the 2015 ISSTA*, 2015, pp. 48–59.

[55] G. Yan, J. Lu, Z. Shu, and Y. Kucuk, "Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability," in *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*. IEEE, 2017, pp. 164–175.

[56] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." International Joint Conferences on Artificial Intelligence Organization, 2020, pp. 3283–3290.

Yinxing Xue received the B.E. and M.E. degrees in software engineering from Wuhan University, Wuhan, China, in 2005 and 2007, respectively, and the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2013.,He is currently a pre-tenure Research Professor with the University of Science and Technology of China, Hefei, China, where he has been a Research Professor with the School of Computer Science and Technology since 2018. His research interests include software program analysis, software engineering, and cyber security issues (e.g., malware detection, intrusion detection, and vulnerability detection).

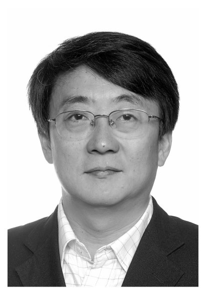
Jiaming YE received the B.E. degree in the Northeastern University in 2013, majored in material science and M.E. degree in the University of Science and Technology of China in 2017, majored in software engineering. He is currently pursuing the doctoral degree in computer science from Kyushu University, Japan. His research interests include software program analysis, software testing and smart contract security issues.

Wei Zhang received the B.E. degree in the University of Science and Technology Beijing in 2017, majored in mechanical engineering. And in 2019, he starts to pursue the master degree in computer technology from University of Science and Technology of China. His research interests include fuzzing test and smart contract security issues.

Jun SUN is currently a full professor at School of Computing and Information Systems, Singapore Management University. He received Bachelor and PhD degrees in computing science from National University of Singapore (NUS) in 2002 and 2006. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship. He has been a faculty member since 2010 and was a visiting scholar at MIT from 2011-2012. Jun's research interests include software engineering, cyber-security and formal methods. He is the co-founder of the PAT model checker.

Lei Ma is currently an Associate Professor with shared appointments between (1) University of Alberta, Canada and (2) Kyushu University, Japan. He is also honorably selected as a Canada CIFAR AI Chair and Fellow at Alberta Machine Intelligence Institute (Amii). Previously, he received the B.E. degree from Shanghai Jiao Tong University, Shanghai, China, and the M.E. and Ph.D. degrees from The University of Tokyo, Tokyo, Japan. His recent research centers around the interdisciplinary fields of software engineering (SE) and trustworthy artificial intelligence (AI) with a special focus on the quality, reliability, safety and security assurance of machine learning and AI Systems. Many of his works were published in top-tier software engineering, AI, and security venues (e.g., TSE, TOSEM, EMSE, ICSE, FSE, ASE, ISSTA, CAV, TNNLS, ICML, NeurIPS, ACM MM, AAAI, IJCAI, TDSC). He is a recipient of more than 10 prestigious academic awards, including three ACM SIGSOFT Distinguished Paper Awards.

Haijun Wang received the Ph.D. degree in the School of Electronic and Information from Xi'an Jiaotong University, China, in 2016. He has worked as post-doctoral research in school of computer science and engineering of Nanyang Technological University. His research interests include program analysis, regression testing, reverse engineering, software security, blockchain security.



Jianjun Zhao received a B.S. degree from Tsinghua University, China, in 1987 and a Ph.D. degree from Kyushu University, Japan, in 1997, both in computer science. He joined the Department of Computer Science and Engineering at Fukuoka Institute of Technology, Japan, as an Assistant Professor and was then promoted to Associate Professor in 2000. From 2005 to 2016, he worked as a Professor in the School of Software and the Department of Computer Science and Engineering at Shanghai Jiao Tong University. Since 2016, he has joined the School of Information Science and Electrical Engineering at Kyushu University as a Professor. His research interests include program analysis and verification, quality assurance for AI systems, automatic programming, software testing, and programming language design.