A Study on Adapting Software Engineering Techniques to New Testing Context with Data-Driven Approach

葉, 家鳴

Adapting Software Engineering Technique to New Testing Context with Data-driven Approach

by

Jiaming Ye

The thesis is submitted to Kyushu University in fulfillment of the requirements for the degree of

Doctor of Philosophy

Graduate School of Information Science and Electrical Engineering

Kyushu University

Under the supervision of:
Professor Jianjun Zhao

July 2023

CONTENTS

Abstract **iv**

Acknowledgements **vi**

List of Figures **viii**

List of Tables **x**

List of Algorithms **xi**

List of Listings **xii**

Publications **xiii**

1 Introduction **1**
- 1.1 Motivation 2
- 1.2 Problem Definition 3
- 1.3 Thesis Approach 5
- 1.4 Thesis Structure 6

2 Background and Related Work **7**
- 2.1 Background 7
  - 2.1.1 GUI Region Detection 7
  - 2.1.2 Well-known Smart Contract Vulnerabilities 9
  - 2.1.3 Cross-contract Vulnerability in Smart Contract 12
- 2.2 Related Work 17
  - 2.2.1 GUI testing and game testing 17
  - 2.2.2 Object detection deep learning models. 18
  - 2.2.3 Smart Contract Vulnerability Detection 18
  - 2.2.4 Smart Contract Testing 20

3 Study on applying data-driven approaches in mobile GUI testing 22
3.1 Introduction 22
3.2 Industry Survey 25
3.2.1 Interview of Mobile Game Testing Experts 25
3.2.2 Questionnaire 28
3.3 Collection of Game GUIs 30
3.3.1 Obtaining Game GUI Dataset 31
3.3.2 Data Cleaning 31
3.4 Evaluation 36
3.4.1 Experiment Preparation 36
3.4.2 Effectiveness of Filtering 38
3.4.3 Model Performance 41
3.4.4 Root Cause Analysis 44
3.4.5 Summary of Findings 46
3.4.6 Threats to Validity 48
3.5 Chapter Conclusion 48

4 Study on vulnerabilities and existing tools in smart contract 49
4.1 Introduction 49
4.2 Overview 52
4.3 Empirical Study of Signature Abstraction 54
4.3.1 Selected Scanners and Dataset 54
4.3.2 Vulnerability Rule Abstraction 56
4.3.3 Case Study: Abstracted Signatures 62
4.3.4 Vulnerability Detection 67
4.4 Evaluation 68
4.4.1 RQ1: Evaluating the Precision of Tools 69
4.4.2 RQ2: Evaluating the Recall of Tools 71
4.4.3 RQ3: Evaluating the Efficiency 72
4.4.4 Threats to Validity 74
4.5 Discussions 75
4.5.1 The Relaxed Security Assumption 75
4.5.2 The Weakness of Vulpedia 75
4.6 Chapter Conclusion 76

5 Study on adapting data-driven methods in smart contract testing 77
5.1 Introduction 77
5.2 Overview 82
5.2.1 Machine Learning Model Training Phase 82
5.2.2 Guided Testing Phase 83
5.3 Machine Learning Guidance Preparation 83
5.3.1 Data Collection 84

5.3.2 Feature Engineering 85
5.3.3 Model Selection 86
5.3.4 Model Robustness Evaluation 88
5.4 Guided Cross-contract Fuzzing 89
5.4.1 Guidance Algorithm 89
5.4.2 Priority Score 92
5.4.3 Cross-contract Fuzzing 94
5.5 Evaluation 95
5.5.1 Dataset Preparation 96
5.5.2 RQ1: Vulnerability Detection Effectiveness 97
5.5.3 RQ2: The Effectiveness of Guided Testing 100
5.5.4 RQ3: Detection Efficiency 103
5.5.5 RQ4: Real-world Case Studies 104
5.6 Chapter Conclusion 106

6 Conclusion **107**
6.1 Summary and Contributions of the Thesis 107
6.2 Possible Directions for Future Work 108

References

Abstract

The software testing phase plays a critical role in the software development lifecycle, but it is often time-consuming, accounting for approximately 50% of the project's time budget. While software testing aims to verify software compliance with requirements, the adaptiveness of existing testing approaches remains a significant challenge. As software applications become more specific to particular domains, adapting existing techniques to new testing contexts poses difficulties due to domain knowledge requirements, differing test criteria, and technical implementation challenges. This thesis focuses on adapting existing approaches in two new testing contexts: GUI testing and smart contract testing. The research is conducted through three key steps:

1. Data-Driven GUI Testing: Leveraging the advancements in AI techniques, the thesis explores the application of data-driven methods in GUI testing. Object detection models are proposed to detect GUI widgets and aid in generating test scripts. A dataset is created using game GUIs for training models, and different models are evaluated for their detection precision. The results show that the trained models achieve a precision of 52.9% and a recall 59.1% on the testing dataset. Challenges in GUI detection, such as compactly placed GUIs and style variety, are also identified and discussed.
2. Smart Contract Testing and Vulnerability Detection: The thesis investigates the domain knowledge required for smart contracts, particularly about vulnerabilities that pose security threats. Vulnerability detection tools for smart contracts are evaluated, and based on their findings, the thesis summarizes four vulnerable signatures and six benign signatures. A vulnerability detector called Vulpedia is implemented, outperforming other tools in terms of precision and recall in vulnerability detection. Vulpedia also exhibits superior efficiency, requiring only 883 minutes to detect vulnerabilities compared to the 8,859 minutes needed by Securify.
3. Data-Driven Smart Contract Testing: The thesis addresses the oversight of cross-contract vulnerabilities in existing smart contract testing tools. To improve the efficiency of detecting cross-contract vulnerabilities, data-driven approaches are

proposed to guide fuzzing testing. A vulnerability dataset is collected and used to train models, achieving a remarkable recall rate of 95% and minimal vulnerability misses. The proposed tool, xFuzz, outperforms other tools by identifying 18 cross-contract vulnerabilities, with 15 of them being missed by existing tools. Additionally, xFuzz detects twice as many vulnerabilities as other tools less than 20% of the time.

In summary, this thesis contributes to adapting existing GUI and smart contract testing approaches in two novel testing contexts. The research demonstrates the effectiveness of data-driven methods in GUI testing, addresses the domain gap in smart contract testing through vulnerability detection, and proposes data-driven approaches to detect cross-contract vulnerabilities efficiently. The findings and tools presented in this thesis offer valuable contributions to enhancing software testing practices in evolving application domains.

Acknowledgements

I am grateful to Kyushu University and the Graduate School of Information Science and Electrical Engineering for funding and supporting me throughout this work.

I want to express my sincere thanks to the members of the defense committee: Dr. Tsunenori Mine, Dr. Jianjun Zhao, and Dr. Lei Ma. Your expertise, time, and constructive feedback have been invaluable to me. Thank you all for your dedication and guidance.

I would like to thank my supervisor, Dr. Jianjun Zhao and Dr. Lei Ma, for giving me the opportunity to do my Ph.D. and for their valuable guidance, unwavering support, and championship throughout this journey. Thank you also to Dr. Paolo Arcaini and Dr. Ishikawa Fuyuki for their supervision in NII.

I would also like to express my gratitude to all my collaborators throughout the years for sharing with me their vast knowledge and research expertise in their respective fields. Thank you for teaching me so much about research and the type of academic I want to be.

Extraordinary gratitude goes to all my friends and colleagues in lab 714 and lab 726 for their support, all the nice times we spent together, and all the rants we shared during all these years.

Last but not least, I am grateful to my family for their unconditional love and support. I am also grateful for my pillar of strength, Lulu Liu, for your gentleness that guides me out of darkness time.

Fukuoka, September 2023 *Jiaming Ye*

LIST OF FIGURES

2.1 An example of cross-contract reentrancy vulnerability which is missed by the state-of-art fuzzer, namely sFuzz. The solid boxes represent functions, and the dashed containers denote contracts. Specifically, the function call is denoted by the solid line. Red arrows highlight the cross-contract calls. The blue arrow represents a cross-contract call missed by sFuzz and ContractFuzzer. . 14

3.1 The overview of the workflow of our survey. 26
3.2 The workflow of building game dataset. 30
3.3 The example of applying contour filtering algorithm in GUI. The red dots denote detected text contours. The blue boxes denote boxes being filtered out, and the green boxes denote boxes being reserved. 33
3.4 The examples of GUIs used in training models. 37
3.5 The example of applying fine-grained filtering method to filter out dirty labels in manually labeled dataset. 41
3.6 Distribution of the number of elements per GUI on RICO and game dataset. 44
3.7 A game GUI example of high widget density. The blue boxes denote labels, and the red boxes denote model predictions. The overlooked boxes are pointed by green arrows. 45
3.8 Comparing the number of GUIs categories in a game and an Android application. Followed by examples of heterogeneous GUI elements in this game. 46
3.9 Return widgets in different games. Widgets in different games are separated by solid lines. 46

4.1 The workflow of extracting vulnerability signatures of Vulpedia. 53

4.2 Three similar code blocks of Unexpected Revert that are found in real-world contracts. Based on their tree edit distance, we cluster them together and abstract a graph skeleton from their PDG. The yellow boxes denote function inputs, the blue boxes denote common nodes on PDG, and the white boxes in the dotted box represent different nodes. 57
4.3 The detection workflow of Vulpedia. 67
4.4 Comparing the vulnerabilities only reported by Vulpedia with vulnerabilities reported by other tools. "Our Unique" means those only found by Vulpedia. 71

5.1 The machine learning training phase of xFuzz framework. 81
5.2 The guided fuzzing phase of xFuzz framework. 81
5.3 The P-R Curve of models. The dashed lines represent performance on the training set, while the solid lines represent performance on the validation set. 87
5.4 The cross-contract fuzzing process. 95
5.5 Comparison of reported vulnerabilities between xFuzz and sFuzz regarding reentrancy. 100
5.6 Comparison of reported vulnerabilities between xFuzz and sFuzz regarding delegatecall. 101

LIST OF TABLES

3.1 The interview questions regarding mobile game testing for two experts. Questions marked green and red box consider the usability and compatibility testing of mobile games, respectively. . 26
3.2 The questionnaire questions. Questions in the green box are for usability testing, and questions in the red box are for compatibility testing. 28
3.3 The composition of game GUIs in our dataset. 32
3.4 Game GUI labels after filtering. The labels filtered by coarse filtering are in red cells, and the labels filtered by fine-grained filtering are in green cells. The "ACF" means widgets after coarse filtering, and the "AFF" means widgets after fine-grained filtering. 38
3.5 The performance of models on the filtered dataset. 40
3.6 The performance of models trained on RICO dataset. (IoU larger than 0.8) 42
3.7 The performance of models trained on the game dataset. (IoU larger than 0.8) 42

4.1 The state-of-art tools for Solidity analysis. 54
4.2 Number of collected contracts for each category 56
4.3 The precision performance of three tools Slither, Oyente, and Smartcheck on four vulnerabilities. 56
4.4 Extracted Signatures from Different Vulnerability Categories . . . 62
4.5 Detection rules for each vulnerability 62
4.6 The detection performance for our tool and other existing ones on the 17,770 contracts, where #N refers to the number of detections, P% and R% refer to the precision rate and the recall rate among the number of detections, respectively. Note that P%= (#TP of the tool)/#N, and R%= (#TP of the tool)/ (#TP in the union of all tools). 68

4.7 The time (min.) of vulnerability detection for each scanner on 76,354 and 17,770 contracts. 73

5.1 Vulnerability detection capability of voting static tools. 84
5.2 The seven static features adopted in model training 85
5.3 The performance of evaluated ML models. 88
5.4 The coverage rate (CR) score of ML model on other tools. 89
5.5 Evaluations on Dataset1. The ✔ represents the tool successfully finding a vulnerability in this function. Otherwise, the tool is marked with ✖. 97
5.6 Performance of xFuzz, Clairvoyance (C.V.), ContractFuzzer (C.F.), sFuzz on cross-contract vulnerabilities. 98
5.7 Performance of xFuzz, Clairvoyance, ContractFuzzer, and sFuzz on non-cross-contract evaluations. 99
5.8 The paths reported by xFuzz and sFuzz. The vulnerable paths found by the two tools are counted respectively. 102
5.9 The time cost of each step in fuzzing procedures. 103

LIST OF ALGORITHMS

1 CoarseFiltering(): traversing all bounding boxes and filtering out invalid coordinates. 34
2 FinegrainedFiltering(): traversing all bounding boxes and filtering out irregularly large boxes and empty boxes. 35
3 Contract Clustering and Signature Abstraction Algorithm 59
4 Machine learning guided fuzzing 90
5 Priorization Algorithm 92

LIST OF CODE SNIPPETS

1 An example of a non-vulnerable code. This is misreported as a vulnerability by Slither and Oyente. 11
2 An example of reentrancy vulnerability. 13
3 An example of delegate-call vulnerability. 13
4 An example of tx-origin vulnerability. 15
5 A real case of using SIG3 (a hard-coded address at line 3), an FP of *reentrancy* for Slither. 64
6 A real case of using SIG4 (an execution lock of `reEntered`), an FP DataDepof *reentrancy* for Slither. 64
7 A real FP of Unexpected Revert reported by Smartcheck, where only one account is involved (SIG6). 64
8 A real FP of *self-destruct abusing* by Slither, as `selfdestruct()` is used under two checks at line 2,3 (SIG10). 66
9 A real case of reentrancy. This is a TP for Vulpedia but an FN for Slither, Oyente, and Securify. 73
10 An example of prioritizing paths. 91
11 A real-world reentrancy vulnerability found by xFuzz, in which the vulnerable path relies on internal calls. 104
12 A cross-contract vulnerability found by xFuzz. This contract is used in auditing transactions in the real world. 105
13 Complex path conditions involving multiple variables and values.106

PUBLICATIONS

Journal Papers

[1] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2022

[2] Jiaming Ye, Mingliang Ma, Yun Lin, Lei Ma, Yinxing Xue, and Jianjun Zhao. Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures. *Journal of Systems and Software*, 192:111410, 2022

Conference Papers

[3] Xiongfei Wu, Jiaming Ye, Ke Chen, Xiaofei Xie, Yujing Hu, Ruochen Huang, Lei Ma, and Jianjun Zhao. Widget detection-based testing for industrial mobile games. In *Proceedings of the 45th IEEE International Conference on Software Engineering*, pages 1427–1437, 2023

[4] Jiaming Ye, Ke Chen, Xiaofei Xie, Lei Ma, Ruochen Huang, Yingfeng Chen, Yinxing Xue, and Jianjun Zhao. An empirical study of gui widget detection for industrial mobile games. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1427–1437, 2021

CHAPTER

ONE

INTRODUCTION

Software engineering techniques are pivotal in designing, developing, testing, and maintaining software applications. Among these stages, testing is the most time-intensive, consuming an estimated 50% of a project's time budget [5]. Software testing is crucial for detecting bugs and vulnerabilities, safeguarding against malicious attacks, and ensuring the robustness of applications. These bugs and vulnerabilities can result in significant financial losses; for instance, in 2011, 2,609 bitcoins were stolen due to software vulnerabilities, equating to a current value of 1.5 billion dollars [6]. Additionally, they can lead to breaches in user privacy, as evidenced by the leakage of 147 million personal records in 2017 due to software bugs [7]. Consequently, industry professionals and researchers strive to develop improved testing techniques. However, the evolving context of software testing makes proposing effective techniques challenging. The challenges and technical aspects vary with each new testing context, indicating no one-size-fits-all solution for all software.

In this thesis, I adapt existing software testing techniques in two rising testing contexts: smart contract vulnerability detection and mobile application GUI testing. I investigate the adaptation of existing software engineering techniques to new testing contexts to detect software vulnerabilities. Specifically, I discovered that a data-driven approach can significantly enhance GUI testing.

Consequently, I leveraged the advantages of data-driven methods to improve testing techniques, thereby enabling the detection of more vulnerabilities.

1.1 Motivation

The evolution of software has significantly changed the world. Various software applications facilitate daily life and expand the boundaries of scientific research and industrial production. The software provides researchers with convenient mathematical tools for simulation or data modeling in specific domains. At the same time, in industries, it enables the operation of heavy robotic arms and the manufacture of precision components. Furthermore, in cutting-edge areas such as artificial intelligence (AI), the impact of software on everyday life is substantial. For instance, security companies employ camera monitors equipped with face detection techniques to swiftly locate suspects [8], and the well-known ChatGPT, offers innovative ways to search for information through conversation [9]. Software has become an integral part of daily life and continues to play an increasingly significant role across various domains.

To develop a robust software application, developers must adhere to the software development lifecycle, which includes design, development, testing, and maintenance. Among these stages, software testing is the most time-consuming, consuming an estimated 50% of a project's time budget [5]. Generally, software testing aims at verifying whether the software can satisfy a list of requirements [10]. Developers write test cases, monitor runtime information, and collect results from the software application under test. These steps enable developers to effectively identify bugs or vulnerabilities if the software behaves contrary to expectations. Researchers propose to automate the testing process, particularly test case generation, due to the time-consuming nature of writing test cases. Software testing research aims to detect bugs or vulnerabilities more effectively and efficiently. Over the past decade, several impressive testing approaches have been proposed, such as AFL [11] and libfuzzer [12].

However, a significant challenge in software testing is the adaptability of existing approaches. Initially, most software applications were developed for general purposes. As software has evolved, applications have become more specific, making adaptation difficult. This often becomes a challenge for software testing when software platforms, computer systems, or programming languages vary. Specifically, adapting existing software techniques to new testing contexts is challenging for several reasons: 1) new test contexts may belong to different domains, requiring domain-specific knowledge; 2) new test contexts necessitate different test criteria and metrics; 3) implementing new test contexts presents technical challenges. As the importance of ensuring software quality in new contexts grows, existing techniques must evolve to meet the current needs of practitioners with new tools and approaches. Due to this adaptability gap, while large companies can afford to write multiple versions of test cases for different software platforms, smaller companies and independent teams are limited by manual efforts and can typically only support one platform or system. Therefore, adapting existing software techniques in a new testing context is urgently needed.

1.2 Problem Definition

The general problem addressed in this thesis is the **adaptation of existing software testing techniques to new testing contexts using data-driven approaches**. Specifically, in this thesis, I study adapting existing software techniques in two testing contexts: mobile GUI testing and smart contract vulnerability detection. I first investigate how much data-driven approaches can enhance automated software testing, particularly for GUI testing. Based on this study, we find that data-driven approaches can help improve software testing techniques. Subsequently, I apply my experience to detect smart contract vulnerabilities. I first survey significant vulnerabilities in smart contracts to gain domain knowledge. Finally, I integrate this domain knowledge with data-driven approaches to detect vulnerabilities in smart contracts. The specific problems and questions can be summarized as follows:

1. To avoid GUI bugs, mobile application companies usually hire a team of test developers to write test cases. A test case usually includes the operation sequences of applications, the location of the operated GUI widget, etc. To ensure the quality of test cases, the test developers must be familiar with the applications under test, which may take extra time

and effort for training developers. To reduce these efforts, companies are seeking techniques that can automatically generate test cases. Along with the rapid development of AI techniques, recent researchers propose to use AI techniques to help generate test cases. To address this problem, we propose to use object detection techniques to detect GUI widgets on the screen to assist in generating GUI test cases. However, how can the object detection techniques be applied in specific steps in GUI testing? To how much extent can the object detection techniques help generate GUI test cases? These questions remain unanswered.

2. Powered by Blockchain technique, smart contracts have attracted attention and been applied in various industries, e.g., financial services, supply chains, smart traffic, and IoTs. However, as one of the most popular languages for blockchain applications, smart contracts have also attracted attention from malicious users. By leveraging smart contract vulnerabilities, the attackers have stolen millions of dollars. Such significant financial loss has sounded the alarm for all users. To secure the smart contract software, the researchers have proposed an impressive list of approaches to detect smart contract vulnerabilities. However, most previous approaches are built upon existing software testing approaches in other platforms, which are not well adapted to smart contract software, causing false positives when detecting vulnerabilities. Without the domain knowledge of smart contracts, detecting smart contract vulnerabilities effectively is difficult.

3. One of the biggest domain challenges in testing smart contract software is the large search space of vulnerability. Specifically, in the business context of smart contracts, cross-contract calls are frequently adopted, which lead to interleaved function calls, making the search space of vulnerability larger than software in other platforms. While previous researchers have proposed a number of tools to detect smart contract vulnerabilities, these tools cannot detect cross-contract call vulnerabilities. To detect this vulnerability, it must first use approaches to reduce the search space.

1.3 Thesis Approach

During the Ph.D., I have proposed approaches that adapt software testing techniques in two new test contexts: mobile GUI testing and smart contract vulnerability detection. Specifically, my approach includes the following three steps:

Study on applying data-driven approaches in mobile GUI testing. In this study, we aim at using data-driven approaches to detect widgets of GUIs of mobile applications and to help further generate test scripts for mobile GUIs. Companies are seeking approaches to automate the GUI testing process to reduce the manual cost of writing test scripts. Seeing AI techniques' developments, we propose using object detection models to detect GUI widgets and help generate test scripts. The GUI widget dataset trains the object detection models to detect the GUI widgets effectively. We first combine the GUI widget dataset from previous works to train the models. We then train two state-of-the-art models, RCNN and YOLO, to detect GUI widgets. Finally, we evaluate the effectiveness of the models and collect comments from test developers by interviewing them. This study shows that data-driven approaches can improve test efficiency but may face challenges when adapting to different domains. Domain knowledge is required when adapting existing techniques in a new test context. This study has been included in my publication [4].

Study on vulnerabilities and existing tools in smart contract. In this study, we aim to build a vulnerability benchmark and implement a vulnerability detector to gain domain knowledge for smart contracts. Study on domain knowledge in smart contracts. The vulnerabilities threaten the security of software in smart contracts. Recalling previous cases, the vulnerabilities in smart contracts have led to millions of dollars in financial losses. Therefore, software testing is required to reduce vulnerabilities in smart contracts. However, there are few reports and surveys on the vulnerabilities of smart contracts, making it difficult to adapt existing software testing techniques to smart contracts. To address this, we conduct a study on the vulnerabilities of smart contracts as well as state-of-the-art tools. We summarize the vulnerability patterns based on the detection rules of existing tools. We also use these patterns to build a vulnerability detector. After the study, we become familiar with the domain knowledge of smart contracts, especially on the critical challenge of vulnerability smart contracts. This study has been included in my publication [2].

Study on adapting data-driven methods in smart contract testing. In this study, we aim to use the benchmark we previously built to train machine learning models and then integrate the models in smart contract testing to detect vulnerabilities with large search spaces. In our study of smart contracts, we find that cross-contract vulnerability is overlooked by existing tools but is important to the security of smart contracts. However, detecting cross-contract vulnerability is not easy. Due to the large search space, existing tools are inefficient when detecting this vulnerability. Recalling our experience of applying data-driven approaches to improve test efficiency in GUI testing, we propose to use data-driven approaches to guide fuzzing testing. We first train a model to detect a smart contract that is potentially vulnerable as candidates. We then prioritize the candidates to test the suspicious contracts first. In the evaluation, our approach outperforms others in test effectiveness and efficiency. This study has been included in my publication [1].

1.4 Thesis Structure

The rest of this thesis is structured as follows:

- Chapter 2 describes some background information helpful in understanding this thesis and presents related work in GUI testing and smart contract vulnerability detection.
- Chapter 3 describes our study on applying data-driven approaches in mobile GUI testing.
- Chapter 4 describes our study on vulnerabilities and existing tools in smart contracts.
- Chapter 5 describes our study on adapting data-driven methods in smart contract testing.
- Chapter 6 concludes the thesis and presents directions for future work.

CHAPTER

TWO

BACKGROUND AND RELATED WORK

This chapter provides an overview of the backgrounds of GUI testing, object detection, smart contract vulnerabilities, etc. It first introduces concepts adopted in this thesis and definitions underpinning the rest of this thesis in Section 2.1, and then discusses works related to this thesis in Section 2.2.

2.1 Background

2.1.1 GUI Region Detection

There is a list of previous works [13–17] that contribute to analyzing characteristics of GUIs (e.g., design style, types of marked tags, usage scenario classification). Owing to these works, the understanding of GUI has been deepened. Researchers have recently proposed GUI region detection for mobile apps, which could be facilities for the analysis of many downstream tasks. For example, Liu *et al.* [18] propose to utilize deep learning models to detect vision issues in GUIs and locate their regions of them; Chen *et al.* [19] propose to combine text-based models and non-text-based models together to improve overall performance in detecting regions of GUI widgets. However, an important list of mobile applications, i.e., mobile games, is overlooked by previous works. Due to the diversity of GUI widgets of mobile games, the approaches of previous works may be ineffective.

2.1.1.1 Models for GUI Region Detection

Faster RCNN and YOLOv2 are currently the state-of-the-art deep learning models for GUI widget detection of mobile apps, which are widely adopted in recent studies [18, 19]. Specifically, faster RCNN is a two-stage anchor-box-based deep learning technique for object detection. It adopts a novel region proposal network (RPN) to predict region proposals with a wide range of scales and aspect ratios. RPN accelerates the generating speed of region proposal because it shares full-image convolutional features and a common set of convolutional layers with the detection network. For each box, RPN then computes a score to determine whether it contains an object and regresses it to fit the actual bounding box of the contained object.

YOLOv2 is a one-stage anchor-box-based object detection technique. It treats GUI widget detection as a regression problem and extracts features from input images as a unified architecture. Different from the manually defined anchor box of Faster-RCNN, YOLOv2 uses the k-means method to cluster the ground truth bounding boxes in the training dataset and takes the box scale and aspect ratio of the k centroids as the anchor boxes. For each grid of the feature map, it generates a set of bounding boxes. For each box, it regresses the box coordinates and classifies the object in the bounding box.

2.1.1.2 Metrics for GUI Region Detection

IoU (Intersection over Union) [20] is an important metric to measure the performance of GUI widget detection. The IoU threshold indicates the requirement for the precision of prediction. The selection of the IoU threshold largely affects the performance of deep learning models. Considering the effect on downstream applications (e.g., widget detection guided GUI testing), a proper criterion to evaluate the performance of models is required.

2.1.2 Well-known Smart Contract Vulnerabilities

In this section, we introduce the four vulnerability types (i.e., `Reentrancy`, `The abuse of tx.origin`, `Unexpected Revert` and `Self-destruct Abusing`.) targeted by our study. The four vulnerabilities deeply threaten the safety of transactions of smart contracts. For example, the `Reentrancy` caused the DAO attack in 2016 and resulted in hundreds of millions of dollars in losses; The `tx.origin` and `Unexpected Revert` vulnerability are listed in the Decentralized Application Security Project (DASP) [21]; The `Self-destruct Abusing` vulnerability often appears with the use of `selfdestruct` instruction in Solidity, and is prone to be exploited if it is not well protected. To motivate this work, we also show a real-world case not well-handled by state-of-the-art scanners.

2.1.2.1 Vulnerability Types

1. *Reentrancy (RE)* As the most famous Ethereum vulnerability, reentrancy recursively triggers the fall-back function [22] to steal money from the victim's balance or deplete the gas of the victim. Reentrancy occurs when external callers manage to invoke the callee contract before the execution of the original call is finished, and it was mostly caused by the improper usages of the function `withdraw()` and `call.value(amount)()`. It was also reported in [23].

2. *The Abuse of `tx.origin` (TX)* When the visibility is improperly set for some key functions (e.g., some sensitive functions with `public` modifier), the extra permission control then matters. However, issues can arise when contracts use the deprecated `tx.origin` (especially, `tx.origin==owner`) to validate callers for permission control. It is relevant to the *access control* vulnerability in [21]. When a user U calls a malicious contract A, who intends to forward a call to contract B. Contract B relies on vulnerable identity check (e.g., `require(tx.origin == owner)` 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.

3. *Unexpected Revert (UR)* In a smart contract, some operations may fail. This can lead to two main impacts: 1) the gas (i.e., the fee of executing an operation in the Ethereum platform) of the transaction is wasted; 2)

the transaction will be reverted, i.e., the denial of service (DoS). The denial of service attack is also termed "DoS with revert" in [24]. The attacker could deliberately make some operations fail for the purpose of DoS. For example, some functions recursively send ethers to an array of users. If one of these calls fails, the whole transaction will be reverted. An attacker can deliberately fail this transaction to achieve a DoS attack.

4. *Self-destruct Abusing (SD)* This vulnerability allows the attackers to forcibly send Ether without triggering its fall-back function. Usually, the contracts place important logic in the fall-back function or make calculations based on a contract's balance. However, this could be bypassed via the `self-destruct` contract method that allows a user to specify a beneficiary to send any excess ether [24]. That is, a vulnerable contract is prone to be exploited to transfer all money to the attacker's account meanwhile shut down the service.

2.1.2.2 Example Programs

Code 1 is mistakenly alarmed by Slither [25] and Oyente [26]. The function `withdraw` intends to send ethers to the `msg.sender`. It first verifies the identity of the caller at line 2. Then, the function reads the amount of the current balance of the caller at line 3 and sends ethers to the caller by using a Solidity call `call.value()()`. Finally, the function updates the balance of the caller at line 5.

The reason for the false alarm of Slither is that Slither detects reentrancy with the following rule:

$$\boxed{\mathit{DataDep}(\_, var_g) \succ \mathit{Call}(\_, var_g) \succ \mathit{DataDep}(\_, var_g) \Rightarrow \text{reentrancy} \qquad (2.1)}$$

In Equation 2.1, *DataDep*(_,*var_g*) denotes write and read operations to variables; var_g denotes a certain public global variable; $\succ$ denotes the execution order in the control flow; $\mathit{Call}(\_, var_g)$ denotes function call operations. This rule describes a common pattern for Reentrancy vulnerability. Code 1 shows a typical example. var_g is usually a balance account (e.g., `balances[msg.sender]`, line 3 in Code 1). An attacker just needs to create a fallback function that calls `withdraw()`. Once `msg.sender.call.value(amount)()` is executed and transfers the funds, the attacker's fallback function [22] will be triggered and call `withdraw()` (line 1) again. This means the attacker can transfer more funds before `balances[msg.sender]` is reduced to 0. This continues until no *ether* remains or execution reaches the maximum stack size.

However, the pattern in Equation 2.1 usually overestimates real Reentrancy vulnerability. The example in Code 1 is a counter-example because the function `withdraw()` is protected by an identity check at line 2. This statement specifies a precondition for running the `withdraw()` function. Once the precondition is not satisfied, the execution will be aborted. In Code 1, the identity check indicates that the contract calling this `withdraw()` function is limited to its `owner` (i.e., the creator of the contract).

Code 1 An example of a non-vulnerable code. This is misreported as a vulnerability by Slither and Oyente.

```
1  function withdraw() {
2      require(msg.sender == owner);
3      uint256 amount = balances[msg.sender];
4      require(msg.sender.call.value(amount)());
5      balances[msg.sender] = 0;
6  }
```

The reason for the false alarm of Oyente is due to that Oyente detects reentrancy with the following rule:

$$\begin{aligned}(\mathit{DataDep}(\_, var_g) \wedge (gas_{trans} > 2300) \wedge (amt_{bal} > amt_{trans}))\\ \succ \mathit{Call}(\_, var_g) \Rightarrow \text{reentrancy}\end{aligned} \tag{2.2}$$

In Equation 2.2, Oyente requires the gas expense to be less than a certain value. In Solidity programs, each transaction requires an amount of gas to complete in the runtime. $gas_{trans} > 2300$ means the gas used for the transaction must be larger than 2300 (2300 is the least gas expense to conduct a transaction call). $amt_{bal} > amt_{trans}$ means the balance amount must be larger than the transfer amount. Finally, the rule of Oyente requires a call to external functions by *Call* meanwhile sending money. Compared with Equation 2.1 (defined by Slither), Oyente has more constraints for gas and balance value.

Similar to the rule of Slither in Equation 2.1, Equation 2.2 also overestimates the condition where a Reentrancy attack can happen. With the protection by the identity check (i.e., line 2 in Code 1), the execution of function calls conforms to the defined runtime conditions but is already free from the Reentrancy attack.

2.1.3 Cross-contract Vulnerability in Smart Contract

Those mentioned four well-known vulnerabilities are intra-contract vulnerabilities. However, they can be extended to cross-contract vulnerabilities, which previous tools can not detect. 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 study.

2.1.3.1 Cross-contract Vulnerability Definition

In general, smart contracts are compiled into opcodes [27] so 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 [28, 29], 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.

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 study, we define three typical cross-contract vulnerabilities that we focus on, i.e., reentrancy, delegate-call, and tx-origin because they are among the most dangerous ones with urgent testing demands. Specifically, the reentrancy and delegate-call vulnerabilities are highlighted as top risky vulnerabilities in previous works [25, 28]. The tx-origin vulnerability is broadly warned at an earlier research [21, 25].

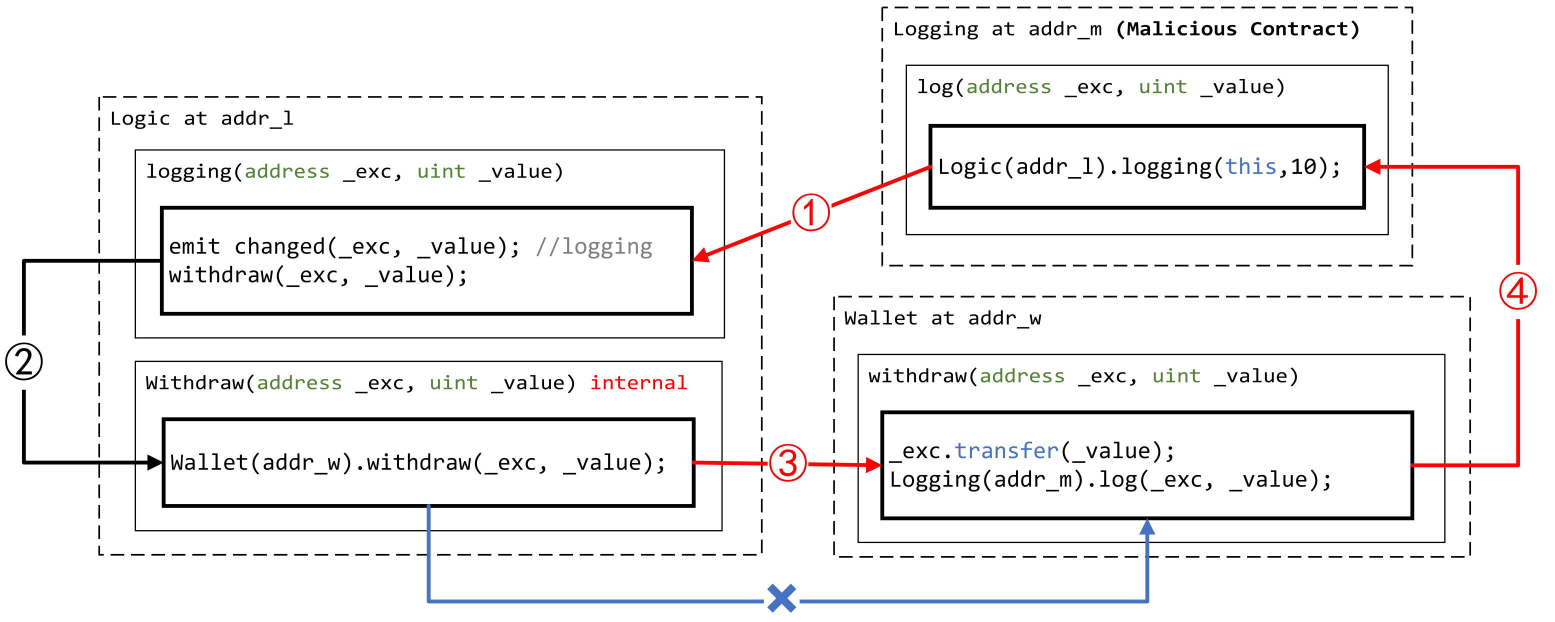
Code 2 An example of reentrancy vulnerability.

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

Code 3 An example of delegate-call vulnerability.

```
contract Delegate {
    address public owner;
    function pwn() {
        owner = msg.sender;
    }  }
contract Delegation {
    address public owner;
    Delegate delegate;
    function() {
      if(delegate.delegatecall(msg.data)) {
        this;
    }  }  }
```

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



```
Logic at addr_l
  logging(address _exc, uint _value)
    emit changed(_exc, _value); //logging
    withdraw(_exc, _value);
  Withdraw(address _exc, uint _value) internal
    Wallet(addr_w).withdraw(_exc, _value);

Logging at addr_m (Malicious Contract)
  log(address _exc, uint _value)
    Logic(addr_l).logging(this,10);

Wallet at addr_w
  withdraw(address _exc, uint _value)
    _exc.transfer(_value);
    Logging(addr_m).log(_exc, _value);
```

Figure 2.1: An example of cross-contract reentrancy vulnerability which is missed by the state-of-art fuzzer, namely sFuzz. The solid boxes represent functions, and the dashed containers denote contracts. Specifically, the function call is denoted by the solid line. Red arrows highlight the cross-contract calls. The blue arrow represents a cross-contract call missed by sFuzz and ContractFuzzer.

Code 4 An example of tx-origin vulnerability.

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

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 steal money recursively. In Code 2, the attacker can construct an end-to-end call chain by calling `withdrawBalance` in the fallback function of the attacker's contract and then steals money.

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

A smart contract suffers from delegate-call vulnerability if and only if at least one of its traces suffers from delegate-call vulnerability. This vulnerability is due to abusing dangerous opcode `DELEGATECALL`. When a malicious attacker B calls contract A by using `delegatecall`, contract A's function is executed in the context of the attacker and thus causes damages. In Code 3, 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 running `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 misusing `tx.origin` to verify access. An example of such vulnerability is shown in Code 4. When a user U calls a malicious contract A, who intends to forward a 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 suffers from cross-contract vulnerability if there is a vulnerable trace (that suffers from reentrancy, delegate-call, 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 2.1. 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 and then sends ethers to the attacker contract by calling function `withdraw` at contract `Wallet`. Next, the wallet contract sends ethers to the 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.

2.1.3.2 The Limitations of the State-of-the-arts

First, we investigate the capability of detecting vulnerabilities by state-of-the-art methods, including [25, 26, 28, 30–32]. Most of these tools, except Clairvoyance, do not support cross-contract testing and analysis. The reason is existing approaches merely focus on one or two contracts, and thus, the sequences and interleavings of a function call from multiple contracts are often ignored. For example, the vulnerability in Figure 2.1 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 a high false positive rate could be well addressed by fuzzing tools by running contracts with generated inputs, existing techniques are limited to a maximum of 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 lead to missed vulnerabilities. To sum up, most existing methods and tools are still limited to handling non-cross-contract vulnerabilities, which motivates this work to bridge such a gap toward solving the currently urgent demands.

2.2 Related Work

This section introduces tools and works closely related to the work presented in this thesis. It first describes GUI testing, game testing, and object detection related to or directly used in this thesis. It then reviews work around vulnerability detection and smart contract testing—the works of all subjects to studies I present in later chapters.

2.2.1 GUI testing and game testing

GUI plays a crucial role in bridging the gap between users and applications. Therefore, previous works have proposed methods to aid GUI development in GUI searching [33–37] based on image features and GUI testing [38–40] based on deep learning models. Specifically, Hu *et al.* [39] proposed automatically generating input cases for GUI testing. They fed the application's input and analyzed the running traces to find bugs. Zhao *et al.* [40] trained a deep learning model to predict workflow actions of applications, which provided valuable experience in applying deep learning to advance the efficiency of GUI tests. However, the above techniques are all developed for general-purpose applications (e.g., hotel booking applications, shopping applications). They do not generalize well for game GUIs.

Games are becoming increasingly popular along with the rapidly developing Internet. One game should be well-tested to eliminate bugs before being published. However, as surveyed by Lin *et al.* [41], even the most popular games are not sufficiently tested. The main reasons for this imperfection can be summarized as the absence of automated testing techniques (i.e., manual testing is still dominant in game testing) due to the survey of Alemm *et al.* [42]. For mobile games, the current works are still preliminary. Lovreto *et al.* [43] developed a method of writing scripts to test functions on 18 mobile games. The limitations of existing techniques are also discussed in their study. Zheng *et al.* [44] first proposed a composite game-testing technique by enhancing reinforcement learning with multi-objective optimization algorithms and outperforming other state-of-the-art.

2.2.2 Object detection deep learning models.

Object detection techniques significantly evolve in the past five years. The mainstream of the proposed methods can be roughly categorized into two-shot detection [45–48] methods and single-shot methods [49, 50], based on their workflows. Object detection models sketch a tight bounding box around the object and classify what the object is. However, the above models are trained to identify objects in the real world. They cannot be directly applied in detecting GUI elements. Deka *et al.* [51] published a dataset with 72K UI screenshots, including widgets, buttons, scrolls, etc. The downloaded screenshots are labeled into 27 categories. Liu *et al.* [52] trained convolutional neural networks on this dataset to detect UI components, which offers us valuable experience in applying deep learning models to detect GUI elements.

2.2.3 Smart Contract Vulnerability Detection

There is already a list of security scanners on smart contracts. From the software analysis perspective, these scanners could be categorized into static- or dynamic-based. In the former category, Slither [25] aims to be the analysis framework that runs a suite of vulnerability detectors. Oyente [26] analyzed the bytecode of the contracts and applies Z3-solver [53] to conduct symbolic executions. Recently, Smartcheck [54] translates Solidity source code into an XML-based IR and defines the XPath-based patterns to find code issues. Securify [28] is proposed to detect the vulnerability via compliance (or violation) patterns to guarantee that certain behaviors are safe (or unsafe, respectively). These static tools usually adopt symbolic execution or verification techniques relevant to Vulpedia. However, none of them applies a code-similarity-based matching technique or considers the possible DMs in code to prevent attacks.

Some other tools enable static analysis for smart contracts. VeriSmart [55] proposes a domain-specific algorithm for verifying smart contracts. VerX [56] combines symbolic execution and contract status abstraction to verify transactions. Zeus [57] adopts XACML as a language to write the safety and fairness properties and converts them into LLVM IR [58] and then feeds them to a verification engine such as SeaHorn [59]. Besides, there is another EVM bytecode decompiling and analysis frame, namely Octopus [60], which needs the users to define the patterns for vulnerability detections. To prevent the DAO, Grossman et al. propose the notion of effectively Callback Free (ECF) objects to allow callbacks without preventing modular reasoning [61]. Maian is presented to detect greedy, prodigal, and suicidal contracts [62], which are different vulnerabilities from the ones we address in this paper. The above tools are relevant, but due to various reasons (e.g., issues in tool availability), we cannot have a direct comparison with them.

The less relevant category includes dynamic testing or fuzzing tools: Manticore [63], Mythril [64], MythX [65], Echidna [66] and Ethracer [67]. sFuzz [31] and Harvey [68] use advanced techniques (e.g., concolic testing, fuzzing, and tainting) for detection. Dynamic tools often target certain vulnerability types and produce results with few FPs. However, they are unsuitable for large-scale detection due to the efficiency issue.

In general, a similar-code matching technique is widely adopted for vulnerability detection. In 2016, VulPecker [69] proposed to apply different code-similarity algorithms for various purposes for different vulnerability types. It leverages vulnerability signatures from National Vulnerability Database (NVD) [70] and applies them to detect 40 vulnerabilities not published in NVD, among which 18 are zero days. As VulPecker works on the source code of C, Bingo [71] can execute on binary code and compare the assembly code via tracelet (partial trace of CFG) extraction [72] and similarity measures. Vuddy [73] targets extracting clones and parameterized clones, not gapped clones, as it utilizes hashing for matching for the purpose of high efficiency. To sum up, these studies usually resort to the vulnerability database of C language for discovering similar zero-days. In contrast, plenty of our efforts are exhausted in gathering vulnerabilities from other tools for smart contracts and auditing them manually. Vulpedia adopts a more robust algorithm (e.g., LCS), which can tolerate big or small code gaps across similar candidates of a vulnerability.

2.2.4 Smart Contract Testing

Our study is closely related to previous works on interactions between multiple contracts. Zhou *et al.* [74] present work to analyze the relevance between smart contract files, which inspires us to focus on cross-contract interactions. He *et al.* [75] report that existing tools fail to exercise functions that can only execute at deeper states. Xue *et al.* [30] studied cross-contract reentrancy vulnerability. They propose constructing ICFG (combining CFGs with call graphs) and then tracking vulnerability by taint analysis.

Our study is also relevant to previous fuzzing work on smart contracts. Smart contract testing plays a vital role in smart contract security. Zou *et al.* [76] report that over 85% of developers intend to do heavy testing when programming. Jiang *et al.* [32] made 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. A similar method is implemented by He *et al.* [75]. They propose to learn fuzzing strategies from the inputs generated by a symbolic expert. The above two methods inspire us to leverage a guide to reduce search space. Tai D *et al.* [31] implement a user-friendly AFL fuzzing tool for smart contracts, based on which we build our fuzzing framework. Unlike these existing works, our work focuses on proposing a 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 outperforms the state-of-the-art in detecting cross-contract vulnerabilities.

The work is also inspired by previous work [77–79]. In their work, they propose learning behavior automata to facilitate vulnerability detection. Zhuang *et al.* [80] offer to build graph networks on smart contracts to extend understanding of malicious attacks. Their work inspires us to introduce machine learning methods for detection. We also improve our model selection by inspiration of work of Liu *et al.* [81]. Their algorithm helps us select the best models with satisfactory performance on recall and precision on highly imbalanced datasets. Yan *et al.* [79] have proposed a method to mimic the cognitive process of human experts. Their work inspires us to find the consensus of vulnerability evaluators to train the machine learning models better.

The smart contract has drawn a number of security concerns since it came into being. As figured out by Zou *et al.* [76], over 75% of developers agree that smart contract software has a much high-security requirement than traditional software. According to [76], the reasons behind such requirements 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 contracts and financial activities, the security of smart contract security largely affects the stability of society.

CHAPTER

THREE

STUDY ON APPLYING DATA-DRIVEN APPROACHES IN MOBILE GUI TESTING

3.1 Introduction

Mobile games are continuously gaining popularity with the advancement of mobile devices over the past decade. According to media [82], the global market share of the game industry is estimated to be more than 85 billion dollars annually, a large portion of which run on mobile devices. Such large potential leads to stiff global competition among game companies. Being supported by modern visualization technology and hardware acceleration of mobile devices, game producers often design mobile games with fabulous and charming visual experiences via graphical user interfaces (i.e., GUIs) to attract more users. Like traditional software, a mobile game can evolve and be updated even more frequently. For example, inside our industrial partner NetEase Games, one of the largest game companies in the world, a typical mobile game usually experiences at least 3 version updates per day for various purposes, e.g., visual/audio feature enhancement, performance optimization, bug fixing, etc. However, the non-trivial amount of update changes can inevitably introduce new bugs that can significantly affect the player experience upon being uncovered by users. Thus, quality assurance of mobile games is of great importance, and systematic testing is often under very tight pressure of frequent version updates.

Although some techniques have been developed for testing the mobile applications, ranging from simple random method Monkey to more advanced methods such as Stoat [83], Sapienz [84], and Espresso [85], they can still be limited for automated mobile game testing due to the unique and highly dynamic characteristics (e.g., heavy user interactions, difficult task accomplishment). Consequently, industrial mobile games still mainly rely on manual testing (i.e., playing games) and semi-automatic testing (i.e., manually written scripts), which are labor-intensive, inefficient, and expensive, becoming the bottleneck of the game testing process for better quality. To this end, some recent works attempted to apply machine learning-based techniques for mobile app testing [39, 40]. However, the challenges and pain points of industrial mobile games are still unclear, and understanding them can be beneficial for better assurance of mobile games.

To bridge this gap, we first conduct a comprehensive survey in *NetEase Games*. In the first step, we performed scrum interviews with two testing experts from the Quality Assurance (QA) department to gain the big picture and better understand the industrial mobile game testing process at the NetEase Games. The interview results show that, in the NetEase Games, the mobile game product is usually tested in terms of the *usability*. The *compatibility*, where *usability* testing aims to detect function bugs and *compatibility* testing ensures that the game can be played smoothly across different devices. Many of these testing tasks are still mainly completed by human testers. To boost the efficiency of mobile game testing, developers or testers are also actively exploring and developing some automated techniques (e.g., POCO [86], monkey [87]). Based on our understanding from the scrum interview, we continue to design a questionnaire to answer **RQ1** — What are the challenges and pain points in industrial mobile game tests? What could be the research opportunities?

Eventually, 50 mobile game testers answered our questionnaire, based on which we identified two main challenges in mobile game testing: a) how to precisely detect the *clickable* GUI widgets of games, especially when the game is deployed on variant end mobile devices, which is of great importance for the following automated testing and analysis tasks and b) how to achieve high coverage especially for the large-scale mobile game with complex logic (e.g., some complicated game tasks). As for the second challenge, some attempts [39, 44, 88] have been made to improve the coverage and detect bugs. While detecting GUI widgets (the first challenge) in mobile games still lacks in-depth investigation, it will be the focus of the rest of this paper.

It was not until recently that machine learning techniques were applied for GUI widgets detection of Android apps. For example, Liu *et al.* [18] propose to utilize deep learning models to detect vision issues in mobile GUIs and locate their regions; Chen *et al.* [19] propose to combine text-based and non-text-based models to improve the overall performance in detecting regions of GUI widgets. However, these techniques are mainly designed and evaluated for conventional mobile apps. Due to the highly dynamic visual effects and interactive nature of game apps, it is still unclear to what extent existing techniques can be adapted to be helpful in the context of mobile games.

Therefore, in this paper, we continue to conduct an empirical study toward understanding the usefulness of existing object detection techniques in detecting GUI widgets of industrial mobile games. To the best of our knowledge, it still lacks a game GUI widget detection benchmark for mobile games that allows systematic study. To further facilitate research along this line, we made significant efforts to construct a mobile game GUI widget detection benchmark. Specifically, we develop an automatic technique to collect game GUI datasets (i.e., screenshots and corresponding widget labels). Since the automatic labeling may introduce inaccurate results, we further adopt a heuristic-based data cleaning strategy to improve the data quality. Finally, we create a game GUI dataset that contains a total of 2,993 GUIs with 38,776 widgets. We then integrated state-of-the-art GUI widget detection methods with our dataset, which together formed the first-ever benchmark to enable the study of GUI widget detection in mobile games.

Based on the constructed benchmark, we performed an empirical study to investigate **RQ2**: How effective are existing object detection methods across various mobile applications? Due to mobile game diversity (i.e., designs), we aim to study how this difference affects model performance. In other words, can the detector trained on some game GUIs be generalized on other games or general-purpose apps?

Furthermore, we performed an in-depth manual analysis to figure out **RQ3**: What are the challenges for detecting GUI widgets in mobile games?

In summary, the contribution of this study is as follows:

- We conduct a survey in the NetEase Games to investigate the urgent challenges and pain points for mobile game testing.
- We develop an automated method to collect and label the game GUI dataset, based on which we create the GUI widget dataset for mobile games. We further integrate state-of-the-art mobile GUI widget detection methods, together forming the first benchmark designed specifically for GUI widget detection research of mobile games.
- We conduct the empirical study to understand better the current status of GUI widget detection of mobile games and to identify challenges and opportunities. We make our benchmark publicly available[1] to enable reproducible study and facilitate further research on downstream tasks such as mobile game testing.

3.2 Industry Survey

To identify the challenges in industrial mobile game testing, in this section, we conduct a two-phase survey on mobile game testing in industries. The workflow is summarized in Figure 3.1. In particular, we first draft an initial set of interviews for gathering the key topics relevant to mobile game testing in industries to deepen our understanding. Next, based on the gathered concerns and questions, we design a structural questionnaire and distribute it to game developers and testers from industries. The questionnaire is mainly designed to understand the pain points in current industrial mobile game testing and further identify the potential opportunities for mobile game testing.

3.2.1 Interview of Mobile Game Testing Experts

In the beginning, to gain the big picture of industrial mobile game testing and understand the testing process in industrial games, we conduct interviews

[1]https://sites.google.com/view/gamedc/

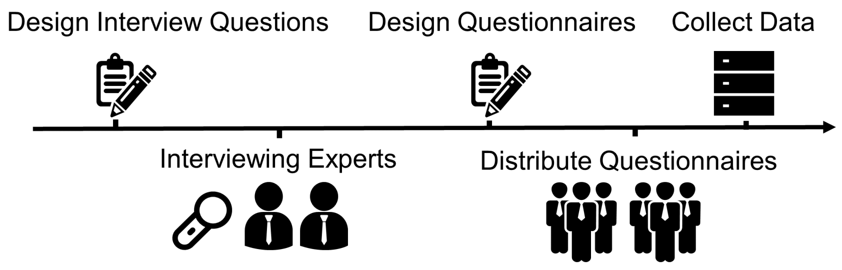


Figure 3.1: The overview of the workflow of our survey.

Table 3.1: The interview questions regarding mobile game testing for two experts. Questions marked green and red box consider the usability and compatibility testing of mobile games, respectively.

ID	Question
I1	Mobile testing routines introduction
I2	Team size
I3	How do we allocate works
I4	Do we cooperate with automated techniques
I5	What are the current limitations
I6	How to ensure high coverage rate by using test case
I7	How much slow down do automated techniques introduce
I8	How does the device difference affect our testing
I9	What is the current alleviation regarding the limitations

with two industry experts who are in the lead position of mobile game testing industries, i.e., the director of the Testing Center (E1) and the leader of the Mobile Testing Lab (E2), respectively.

As part of the formal interview procedure, we have designed nine questions summarized in Table 3.1 to help us better understand the current status of mobile game testing in industries.

When the interview starts, we first invite the interviewee to give a basic introduction about the mobile game testing teams (I1, I2). Then, we ask questions about the testing routines of mobile games (I3) and the current automated techniques adopted in mobile game testing (I4). We also ask about the limitations captured by developers (I5). Then, we prepare specific questions (I6 to I9) for different interviewees based on the mobile testing tasks they mainly lead. For E1, we focus on usability testing and ask about the current solutions and limitations (I6, I7). For E2, we prepare questions about the technical details in compatibility tests (I8, I9). Both interviews are arranged in 30 minutes. We record the interviews and intend to make them publicly available after the internal approval of industries.

Overall, we obtain a valuable high-level understanding from interviews. For example, in compatibility testing, there are around 10 test developers who are responsible for testing device compatibility. They often play the same interaction behaviors on multiple devices. The testing developers mainly adopt OneToMany [87] techniques to achieve that goal. They first record interaction behaviors on one mobile device and then replay the sequence of behaviors on other devices. This technique is somehow efficient but can be limited to device differences, mainly caused by resolution differences. When they replay the interaction behaviors, the action may be replayed at an incorrect location and thus affect the testing efficiency. The current solution is to categorize mobile devices with similar resolutions into one group to eliminate the impact of resolution diversity. However, as the number of groups increases, it is still inefficient and becomes the bottleneck for continuous testing pressure of frequent updates.

In usability testing, more than 50 test developers concentrate on testing mobile game usability tasks. Usability testing of mobile games relies on writing scripts to test the runtime status of games. To ensure the scripts cover most game scenes, experienced developers usually double-check the scripts. To facilitate testing, they often utilize POCO [86] to extract GUI information (e.g., clickable region coordinates) that can help write testing scripts. However, POCO has the following limitations: 1) POCO is limited to game-developing engines (i.e., Unity3D, Cocos2dx), and 2) POCO often misleads the testing by reporting wrong coordinates due to ignoring visual effects. For example, some clickable widgets are overlapped by other widgets. They are actually not clickable for users, but POCO still reports them.

In summary, we gain the big picture of industrial mobile game testing and better understand the testing process from the scrum interviews. Currently, many of these testing tasks are mainly completed by human testers, which

Table 3.2: The questionnaire questions. Questions in the green box are for usability testing, and questions in the red box are for compatibility testing.

ID	Question
Q1	Years of testing experience
Q2	Gender
Q3	Job responsibility
Q4	Averaged number of monthly testing scripts
Q5	How many testing scripts are needed in a project?
Q6	Time for writing a testing script
Q7	Whether use automated techniques to aid testing?
Q8	Challenges during testing
Q9	Averaged number of weekly testing projects
Q10	Whether use automated techniques to aid testing?
Q11	Averaged number of devices in one testing
Q12	Time for particular testing
Q13	Averaged number of groups in compatibility testing
Q14	Dream tools

may lead to inefficiency. To improve this and automate the testing process, developers and testers are also actively exploring and developing automated techniques (e.g., POCO [86], monkey [89]).

3.2.2 Questionnaire

To further reveal the challenges of automated game testing and understand the potential opportunities, we continue to design a fine-grained questionnaire for the developers and testers. The questionnaire is generally based on our interviews but has more detailed questions, as shown in Table 3.2. Questions in white cells are common questions, questions in green cells are for usability testing developers, and questions in red cells are for compatibility testing developers. We prepare specific questions for different testing tasks to discover the in-depth needs of developers. The questionnaires are anonymously distributed to forefront developers in the Testing Center and testers from industries in the Mobile Testing Lab. At last, we received 53 questionnaires. Forty-four are about usability testing, and nine are about compatibility testing.

The results of the questionnaires show that 52% (23 of 44) of the developers are senior ones with more than three years of experience. Among them, 77% (34 of 44) developers adopt semi-automated tools (i.e., POCO) in usability testing to assist in writing testing scripts. We also find that 21 of the 34 developers spend about 9 to 35 hours one-week drafting testing scripts. Additionally, we highlight 27 of the 34 developers complain that 1) POCO often misleads testing by ignoring visual effects (e.g., overlapping, shadowing) and 2) the time cost of using POCO is often unacceptable. We also observe some developers highlight that 1) POCO-based test script is not flexible for maintenance and 2) POCO is not stable that which often leads to program crashes. Finally, 85% (29 of 34) developers are convinced that applying widget detection methods in usability testing can increase by at least 35% of testing speed.

The questionnaire results also indicate that 67% (6 of 9) compatibility testing developers adopt OneToMany [87] technique to assist testing such that the efficiency can be improved. 77% of the developers have more than two years of experience in compatibility testing. Six of them usually work on at least three projects in a week, and 77% of developers (7 of 9) have to test more than eight devices in one testing task. Note that 67% (6 of 9) developers complain that testing procedures suffer from significant differences in the resolutions of devices. The 67% developers alleviate the effect of resolutions by grouping devices in similar resolutions into more than three categories. This alleviation can reduce the resolution difference in one group. However, the workload of testing procedures is increased mainly since it has to be replayed among all groups. The resolution difference makes the compatibility testing inefficient. In our investigation, 77% of them agree that applying widget detection methods in testing can definitely increase the efficiency of the testing process by at least 50%.

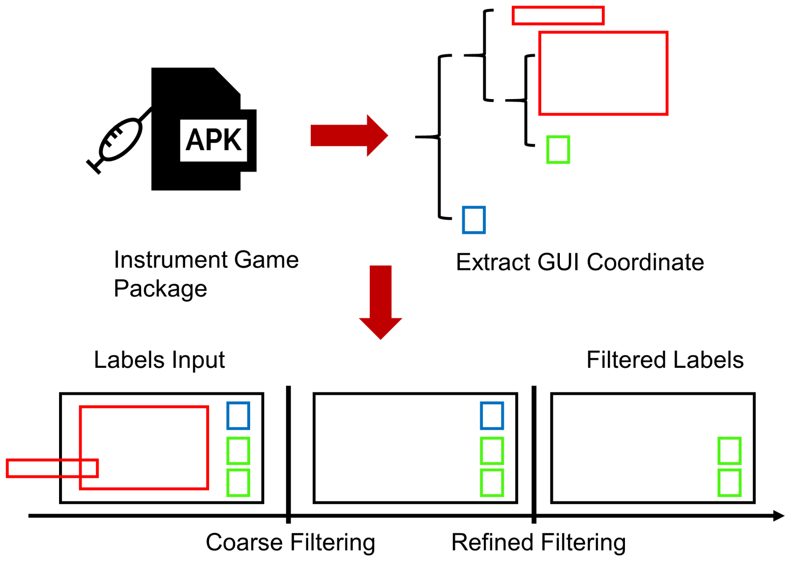


Figure 3.2: The workflow of building game dataset.

Answer to RQ1: From our interviews, we know that existing mobile game testing tasks are mainly completed by human testers, which is very inefficient. Thus, automated testing is urgently needed for industrial games. To this end, developers and testers explore and develop tools to assist the testing. However, these techniques have severe limitations (e.g., inaccurate results, different resolutions). The response to questionnaires indicates that how precisely detect the *clickable* GUI widgets of mobile games is of great importance for automating testing tasks.

3.3 Collection of Game GUIs

To the best of our knowledge, there is no game GUI benchmark now. We plan to construct a mobile game GUI dataset to facilitate mobile game testing further. The overview of our workflow is shown in Figure 3.2. Specifically, we develop an automatic technique to collect game GUI images and label the widgets automatically in subsection 3.3.1. Since the automatic labeling may introduce inaccurate results, we further adopt a heuristic-based data cleaning strategy to improve the data quality in subsection 3.3.2.

3.3.1 Obtaining Game GUI Dataset

We developed a technique that can automatically locate the click widgets and outputs their region information, based on which we manually label the widgets. Specifically, we first instrument the game software by POCO such that, when a game is started, POCO can automatically instantiate and watch global members of game object [90]. After installing the instrumented game on mobile devices, we connect the device to the computer and enable Android ADB to debug mode [91] on mobile devices. The screenshots, as well as the GUI tree, are captured during the game playing. We then extract GUI widget coordinates based on the GUI tree and label the widget. However, there are a number of irrelevant widgets (e.g., widgets that are not clickable). We remove these widgets by filtering their element types (e.g., unclickable widgets often have the element type "Scene"). At last, we obtain 1,135 GUIs with 17,808 elements from four games (i.e., Elysium of Legends, Dream Chaser, Butterfly Swords, and AllStar). Since mobile games are mostly played horizontally, the game screenshots are rotated for 90 degrees to keep the same size as the existing GUI dataset.

To enrich our dataset with games published by other companies, we spent three weeks downloading games, taking screenshots, and manually labeling clickable widgets. We chose five games (i.e., Onmyoji, Arena of Valor, Princess Connect, Naruto, and SevenDay) released by the NetEase Games, *Tencent Games* and *Cygames*. Note that while the two games, Onmyoji and SevenDay, are developed by NetEase Games, they cannot be instrumented by POCO, so we have to label them manually. After enough screenshots of a game (i.e., the screenshots could cover at least 80% of play scenes) are collected, we label the clickable widgets by a third-party labeling tool [92]. We obtain 1,849 GUIs and 20,968 widgets from the above games. The description of the game dataset is shown in Table 3.3. To avoid introducing manual mistakes, all labels are cross-checked by our collaborators.

3.3.2 Data Cleaning

After collecting the 2,993 GUIs with 38,776 widgets, we observe that some inaccurate labels may decrease the performance of models. Generally, inac-

Table 3.3: The composition of game GUIs in our dataset.

Game Name	#N of GUIs	#N of Widgets	Released by
Elysium of Legends	503	6,873	NetEase Games
Dream Chaser	224	3,838	NetEase Games
Butterfly Swords	307	5,160	NetEase Games
AllStar	110	1,937	NetEase Games
Onmyoji	693	8,050	NetEase Games
Arena of Valor	183	2,856	Tencent Games
Princess Connect	137	1,268	Cygames
Naruto	383	4,138	Tencent Games
SevenDay	453	4,656	NetEase Games
Total	2,993	38,776	

curate labels can be grouped into two categories: (1) invalid labels and (2) incorrect labels.

The invalid labels include labels with negative coordinates and labels with coordinates outside the screen. These labels are mainly due to the limitation of the automatic labeling technique by extracting GUI information with POCO. Specifically, POCO automatically extracts GUI trees from screens. Some illegally placed widgets (i.e., placed with negative coordinates or placed outside the screen) are included in labels without filtering. These invalid labels cause errors during training models. Therefore, they need to be cleaned before training.

The incorrect label denotes labels valid for training but harmful for training models. The incorrect label consists of two categories: (1) the irregularly large boxes and (2) empty boxes. Specifically, the irregularly large box denotes that the size of a box is far larger than a regular box; these boxes produced by POCO development kit are ineffective for locating a clickable button. The empty boxes are incorrectly labeled by POCO due to the unawareness of visual overlaps among widgets. For example, in Figure 3.3, the two blue boxes are widgets in previous screens. They are buried under the current background and cannot be clicked. However, they are detected by POCO and are

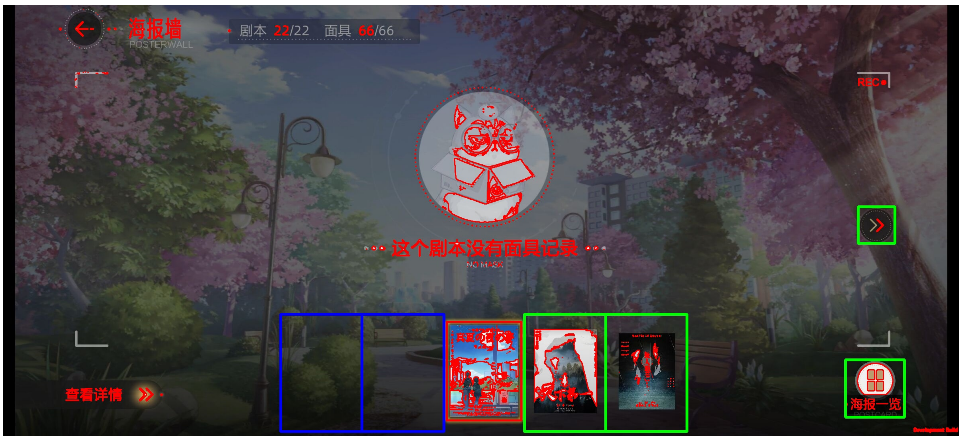


Figure 3.3: The example of applying contour filtering algorithm in GUI. The red dots denote detected text contours. The blue boxes denote boxes being filtered out, and the green boxes denote boxes being reserved.

mislabeled.

To filter out such invalid and incorrect labels, we propose a label filtering method to remove dirty labels. Our method includes coarse filtering and fine-grained filtering.

Coarse filtering. We design the coarse filtering in Algorithm 1. The coarse filtering mainly focuses on invalid coordinates (i.e., with negative values or coordinates out of the screen). The input of Algorithm 1 includes the image source P and corresponding bounding boxes B. The output is a set of valid boxes. Algorithm 1 is composed of four steps: (1) obtaining the width and height of the input screenshot at line 2; (2) checking if the coordinates of a box contain negative values at line 8; (3) checking if the coordinates are outside the screen at line 10 by comparing the width and height of the screen with the coordinates; (4) finally outputting a valid set of boxes of the screenshot. If the input set has n screenshots and m bounding boxes, the time complexity of line 2 is $O(nm)$.

Fine-grained filtering. Fine-grained filtering aims at filtering out irregularly large bounding boxes and empty boxes.

To remove the irregularly large bounding boxes, we propose to filter them by comparing the size of the boxes with the size of the screen. That is, given a threshold factor λ, a bounding box whose size is larger than λ percent of

Algorithm 1: CoarseFiltering(): traversing all bounding boxes and filtering out invalid coordinates.

```
input  : $P$, all the screenshots
input  : $B$, all the bounding boxes
output: $VB \leftarrow \emptyset$, the set of valid boxes
foreach screenshot $p \in P$ do
    $w, h \leftarrow p.getPitureSize()$
    //get screenshot width and height
    foreach bounding box $b \in B$ do
        $VB \leftarrow VB \cup \{b\}$
        $c \leftarrow b.getCoordinates()$
        // get values of the rectangle points
        if $hasNegtiveValues(c)$ is True then
            $VB \leftarrow VB - \{b\}$
        if $hasOutsideScreenCoors(c, w, h)$ is True then
            $VB \leftarrow VB - \{b\}$
return $VB$
```

Algorithm 2: FinegrainedFiltering(): traversing all bounding boxes and filtering out irregularly large boxes and empty boxes.

```
input  : $P$, all the picture source
input  : $B$, all the bounding boxes
input  : $\lambda$, the factor for determining the size of large boxes
output: $VB \leftarrow \emptyset$, the set of valid boxes
1  foreach picture $p \in P$ do
2      $w, h \leftarrow p.getPitureSize()$
3      $S \leftarrow w \times h$
4      // calculate the size of screen
5      $Con \leftarrow p.findAllContours()$
6      // find contours by Suzuki's Contour tracing algorithm
7      foreach bounding box $b \in B$ do
8          $VB \leftarrow VB \cup \{b\}$
9          $S_b \leftarrow b.getBoxSize()$
10         // calculate the size of bounding box
11         if $S_b > \lambda \times S$ is True then
12             $VB \leftarrow VB - \{b\}$
13         foreach contour $c \in Con$ do
14             if $b.contain(contour)$ is True then
15                 break;
16             $VB \leftarrow VB - \{b\}$
17 return $VB$
```

the size of the picture will be dropped. To find a proper value for λ, we randomly pick 10,134 bounding boxes from the game dataset and collect the size of these boxes. Next, we compare the size of the boxes with the size of the screen. We find that 99.6% of bounding boxes have less than 10% of the size of the screen, which means that almost all bounding boxes are small in games. As we further investigate the 0.4% large boxes, we find they are all unrelated background widgets and are not clickable. Thus they should be removed. Implementing this algorithm sets the factor λ to 10.

To filter empty boxes, we apply Suzuki's Contour Tracing Algorithm [93] that reports points of textures, text, and contours of widgets. We empirically find that most empty boxes contain no contours. Therefore, we remove boxes that have no points. For example, in Figure 3.3, the reported points from our algorithm are denoted as red dots. As the blue boxes contain no red dots, these boxes are considered empty and removed from the labels. Differently, all of the green boxes contain red dots, and they are deemed valid labels.

Algorithm 2 shows the method that is composed of four steps: (1) getting the size of the input image at line 2; (2) finding all contours in the picture by using Suzuki's Contour tracing algorithm at line 5; (3) calculating the size of bounding box and compare it with image size at line 9 to line 12; (4) checking if there exists a contour point within the bounding box at line 13 to line 16. If the input set has in total n images, m bounding boxes, and we find c contours in a picture, the time complexity of Algorithm 2 is $O(nmc)$.

3.4 Evaluation

3.4.1 Experiment Preparation

3.4.1.1 Dataset

To adopt existing methods [18, 19], we follow previous works [19] to download and preprocess the well-known RICO [51] dataset. The RICO GUI dataset contains 66,261 GUI screenshots and 199,830 GUI widgets. We remove widgets that are not clickable (e.g., text widgets, images) and not visible to users (e.g., overlapped widgets, shadowed widgets). After this, we obtain 61,906 widgets.

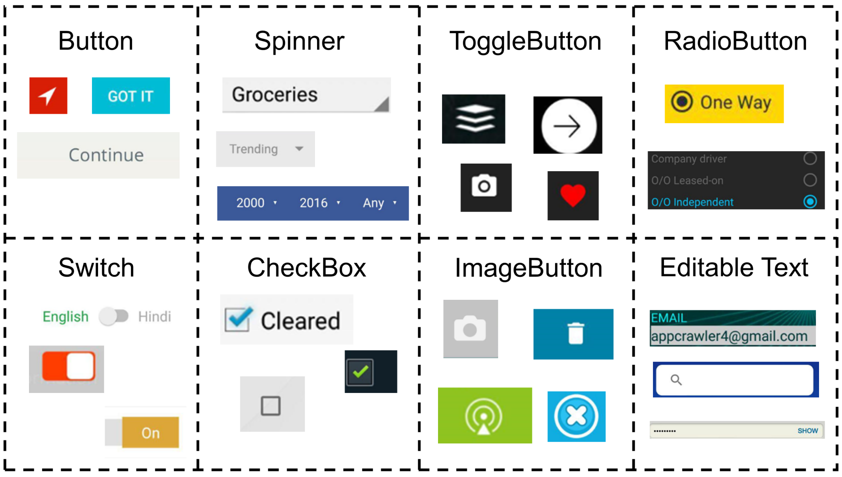


Figure 3.4: The examples of GUIs used in training models.

We further filtered out 31,985 GUI widgets with incorrect coordinates (e.g., coordinates outside the screen). Finally, we collected 30,011 widgets. Examples of widgets used in our dataset can be found in Figure 3.4. Since the GUI screenshots have different image sizes, we resize them to the fixed resolution of 1440*2560. We split 30,011 elements RICO dataset into train/validation/test dataset with a ratio of 8:1:1 (24K:3K:3K). For the game dataset, we use the data that is automatically labeled as the training set and use the data that is manually labeled as the testing set because the manual labels are more accurate for the evaluation.

3.4.1.2 Model Training

We follow the training method in previous works [18, 19]. For Faster RCNN and YOLOv2, our training is based on their models, which are pre-trained on the COCO object detection dataset. We keep the default batch size of 256 and use SGD optimizer for Faster RCNN. Meanwhile, we use the default batch size of 64 and use Adam optimizer for YOLOv2. Faster RCNN uses VGG16 [94] as the backbone, while YOLOv2 utilizes Darknet19 [95] as the backbone. Both models are trained for 45,000 iterations to ensure they are sufficiently trained. As the models may predict duplicated bounding boxes regarding the same object, we use non-maximum suppression (NMS) to remove redundant boxes and keep the best one.

3.4.1.3 Metrics

We adopt the metrics used in previous works [18, 19] to evaluate the performance of our models. Specifically, we use the precision, recall, and F1 scores to evaluate the performance of models. The intersection over union (IoU) indicates the intersection of two regions, A and B, divided by the union region of them. The IoU threshold affects the precision of prediction. In previous works, the threshold ranges from 0.3 to 0.9. In order to better evaluate the model precision (i.e., the IoU threshold is not too low or not too high), we set the IoU threshold as a comparatively strict value of 0.8 in our experiments. A true positive prediction (TP) is the prediction that satisfies both the confidence threshold and IoU threshold, while a false positive prediction (FP) is the prediction that only satisfies the confidence threshold. The false negative (FN) is the region missed by models. We calculate the precision rate by $TP/(TP + FP)$ and the recall rate by $TP/(TP + FN)$. We further compute the F1 score as: $F1 = (2 \times Precision \times Recall)/(Precision + Recall)$.

3.4.2 Effectiveness of Filtering

Table 3.4: Game GUI labels after filtering. The labels filtered by coarse filtering are in red cells, and the labels filtered by fine-grained filtering are in green cells. The "ACF" means widgets after coarse filtering, and the "AFF" means widgets after fine-grained filtering.

	Elysium of Legends	Dream Chaser	Butter Swords	AllStar
Original	6,873	3,838	5,160	1,937
Negative	639	0	0	0
Outimg	976	1,187	37	34
ACF	5,258	2,651	5,123	1,903
Large label	189	103	18	26
Empty label	112	546	1,159	391
AFF	4,957	2,002	3,946	1,486

3.4.2.1 The Filtered Dataset

The labels filtered by coarse filtering and fine-grained filtering are shown in Table 3.4. In this table, “negative” and “outimg” denote coordinates that have negative values and coordinates outside the screenshot. The “large box” represents the bounding boxes with irregularly large sizes, and the “empty box” represents the boxes containing no widgets.

We observe that only one game (i.e., Elysium of Legends) has negative coordinates, and four games (i.e., Elysium of Legends, Dream Chaser, Butterfly Swords, AllStar) have incorrect coordinates outside the screen. The reason is that the labels in these four games are automatically generated by our GUI information extraction technique (See subsection 3.3.1). Since the extracted widgets are filtered by rules, some widgets with the out-of-GUI coordinates may be missed by rules and are included in our dataset. The model training cannot start when these invalid coordinates exist. Therefore, we leverage coarse filtering to eliminate labels with invalid coordinates. After coarse filtering, 639 coordinates with negative values and 2,234 coordinates out of the screen are removed.

The fine-grained filtering filters 336 irregularly large boxes. We observe that the games that are automatically labeled (i.e., Elysium of Legends, Dream Chaser, Butterfly Swords, and AllStar) tend to contain more dirty labels (e.g., 189 in Elysium of Legends and 103 in Dream Chaser). Our method also helps filter out empty boxes, and the results show that most game labels contain a number of empty boxes. Recall that the empty boxes are ones that are incorrectly labeled by POCO due to the unawareness of visual overlaps among widgets. For example, in Figure 3.3, the blue boxes are widgets in previous screens. They are buried under the background and not clickable on the current screen.

3.4.2.2 Model Performance on Filtered Dataset

In the evaluation, we use the games that are automatically labeled (i.e., Elysium of Legends, Dream Chaser, Butterfly Swords, and AllStar) as training datasets, and the manually labeled games (i.e., Onmyoji, Arena of Valor, Princess Connect, Naruto, SevenDay) as testing dataset. The results are shown

Table 3.5: The performance of models on the filtered dataset.

	Before			After		
Setting	P	R	F1	P	R	F1
FasterRCNN Default	12.9%	11.7%	12.2%	17.3%	16.4%	16.9%
FasterRCNN Customized	0.5%	0.1%	0.1%	0.8%	0.2%	0.3%
YOLOv2 k=5	0.3%	≈0%	0.1%	7.6%	5.5%	6.4%
YOLOv2 k=9	4.4%	2.3%	3.0%	6.5%	5.0%	5.7%

in Table 3.5. In this table, the "Before Filtering" denotes the performance of models on the dataset only with coarse filtering, and the "After Filtering" denotes the performance of models on the dataset, which has been processed by fine-grained filtering. Note that the dataset used here is already processed by coarse filtering because negative or out-of-GUI coordinates can largely affect the performance. Hence, we mainly evaluate the model performance before/after the fine-grained filtering.

We observe that the performance of all models increases after we apply the fine-grained filtering. Specifically, Faster RCNN with default settings has the best performance with the highest precision (17.3%) and recall (16.4%) on the filtered dataset. Compared with the model with the same settings but on the unfiltered dataset, the improvement of the dataset leads to around 4.6%, 4.6%, and 4.7% increase in precision, recall, and F1 score, respectively. Faster RCNN with customized settings keeps the same unsatisfactory performance as in the previous empirical study. The performance of this model slightly increases on the filtered dataset. YOLO with k value 5 obtains the largest improvement on the filtered dataset. The processed dataset leads to more than 7% improvement in precision and at least 5% improvement in recall and F1 score. Comparatively, the performance of YOLO with k value 9 is improved on the filtered dataset for at least 2%. In summary, the filtered dataset effectively improves the model performance in terms of precision, recall, and F1.

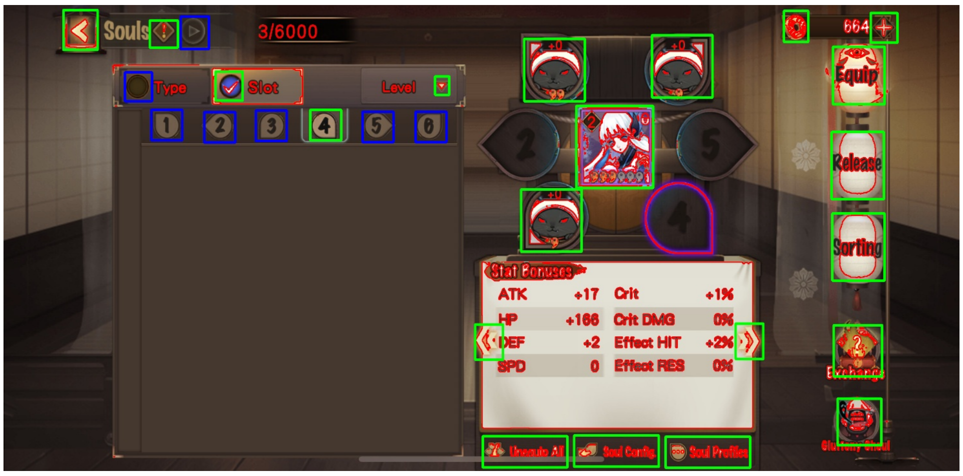


Figure 3.5: The example of applying fine-grained filtering method to filter out dirty labels in manually labeled dataset.

3.4.3 Model Performance

In this section, we conduct experiments to evaluate the model performance on the RICO and game dataset. For models adopted in our experiments (i.e., Faster RCNN and YOLOv2), we prepare two sets of hyper-parameters to evaluate their performance. Specifically, for Faster RCNN, we change the original three anchor scales (8, 16, 32) to larger scales (16, 32, 64). We make this change in consideration that larger anchor scales may better fit the objects in GUIs. The Faster RCNN with larger scales is represented by "Faster RCNN Customized". For YOLOv2, we use two sets of k values (i.e., 5 and 9) in consideration that more anchors may improve the model performance. The two values are also adopted in previous works [49] for evaluations. The dataset for our evaluations includes RICO and the game dataset.

3.4.3.1 Model Performance on Game Dataset

We compare the performance of deep learning models on both RICO and the game dataset. We train our models with different settings on RICO dataset and test our models on the game dataset. The results are listed in Table 3.6 and Table 3.7. The "P" and "R" in this table denote the precision score and recall score, respectively. The "FR" represents the deep learning model Faster RCNN and the "FR Customize" means the model with changed anchor sizes. We also

Table 3.6: The performance of models trained on RICO dataset. (IoU larger than 0.8)

	On RICO			On Game		
Setting	P	R	F1	P	R	F1
FasterRCNN Default	52.9%	59.1%	55.9%	23.4%	4.6%	7.7%
FasterRCNN Customized	22.9%	10.4%	14.3%	1.1%	0.2%	0.3%
YOLOv2 k=5	50.7%	37.9%	43.4%	9.8%	1.3%	2.3%
YOLOv2 k=9	49.5%	36.5%	42.0%	10.0%	1.2%	2.2%

Table 3.7: The performance of models trained on the game dataset. (IoU larger than 0.8)

	On RICO			On Game Dataset		
Setting	P	R	F1	P	R	F1
FasterRCNN Default	4.1%	3.7%	3.9%	12.9%	11.7%	12.2%
FasterRCNN Customized	0.1%	0.1%	0.1%	0.5%	0.1%	0.1%
YOLOv2 k=5	≈0.0%	≈0.0%	≈0.0%	0.3%	≈0.0%	0.1%
YOLOv2 k=9	≈0.0%	≈0.0%	≈0.0%	4.4%	2.3%	3.0%

evaluate the YOLOv2 model with different *k* values to study the impact of the parameters on model performance.

We observe that the precision and recall of models drop sharply on the game dataset. Faster RCNN has 55.9% F1 score on RICO dataset but only 7.7% on the game dataset. Specifically, the precision of Faster RCNN drops half on games, and the recall drop from 59.1% to 4.6%. A recall under 5% is rather low for a model and denotes that the model is ineffectively to cover most GUI widgets. We also observe that the Faster RCNN with customized settings performs worse than the model with default settings and perform even worse on game datasets with an F1 score of 0.3%. Recall that we attempt to change the anchor size and expect the model to precisely match bounding box labels. However, the experiment results show that our configuration with a larger anchor size leads to worse performance.

YOLO models perform close to Faster RCNN regarding the precision of 50.7%. However, the recall of YOLO is far worse than Faster RCNN. We observe that the performance of YOLO also sharply drops when tested on the game dataset, with an F1 score decreasing from 43.4% to 2.3%. Recall that we change the model setting of YOLO to a larger value of *k* (i.e., 9) and expect the YOLO model to generate more anchors to match bounding box labels. The experiments show that the configuration of a larger of *k* results in a worse performance.

3.4.3.2 Model Performance Across Various Mobile Applications

We also train models on the game dataset and evaluate their performance on RICO and the game dataset, respectively. The models are with the same settings as models trained on RICO. The results are shown in Table 3.6 and Table 3.7. We observe that the models perform much worse on the test dataset. Specifically, comparing the models tested on the game and RICO dataset, their performance drops sharply. For Faster RCNN, the precision rate on the test dataset drops from 12.9% to 4.1%, and the recall rate decreases from 11.7% to 3.7%. The performance of RCNN with larger anchor sizes (i.e., customized settings) also drops from 0.5% to 0.1%. The Faster RCNN with the default setting performs better transferability on different datasets. For YOLO models, the performance of models with the two settings (k= 5, 9) drops to less

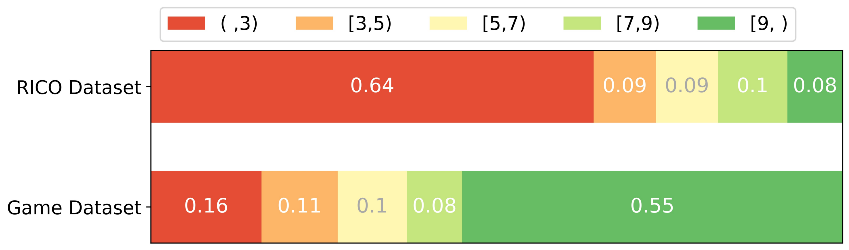


Figure 3.6: Distribution of the number of elements per GUI on RICO and game dataset.

than 1% regarding precision, recall, and F1 score. The YOLO models fail to produce an effective prediction on different datasets.

3.4.4 Root Cause Analysis

3.4.4.1 Game GUI Element Density

We first take a deep look at the UI density difference between regular application GUIs from RICO dataset and game GUIs from our collected dataset. The basic distribution of GUI density is summarized at Figure 3.6. We observe that 55% of game GUIs have more than 9 elements, and in contrast, only 8% of GUIs in regular applications contain more than 9 widgets. Most regular applications contain no more than three widgets in one interaction screen. Differently, game applications intend to offer more interaction options to attract users. As the game GUI elements are often placed side by side and separated by only small padding, detecting GUI regions under high density is challenging for models. For example, Figure 3.7 is a screenshot of character selection. There are 50 elements in this GUI, and each element is placed close to the others. Our models predict most regions of these elements but missed 5 boxes.

3.4.4.2 Art Style Diversity in Game

The game GUIs are subject to art design needs. In fact, in some typical kinds of games (e.g., Massive Multiplayer Online Role-Playing Games), the mechanics of gameplay are rather complex. Unlike regular applications aiming at

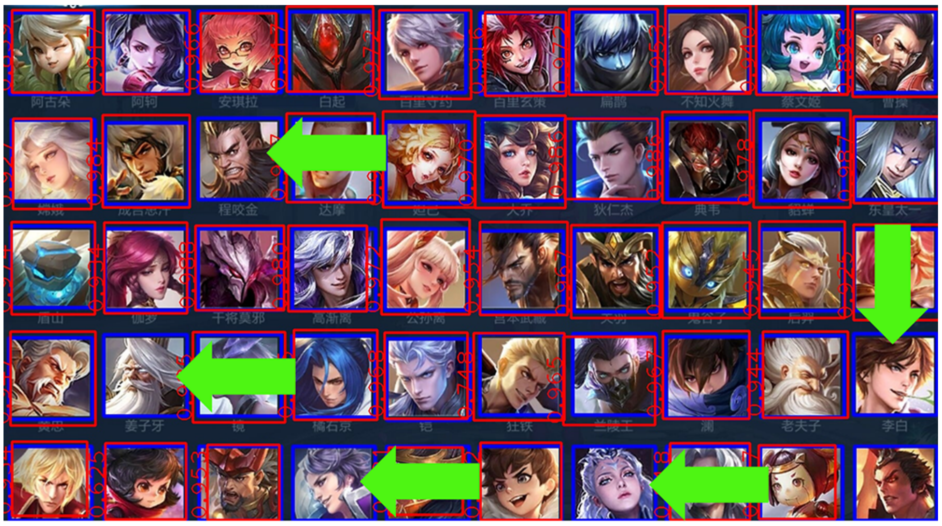


Figure 3.7: A game GUI example of high widget density. The blue boxes denote labels, and the red boxes denote model predictions. The overlooked boxes are pointed by green arrows.

providing a series of services, game developers intend to build a virtual world to attract users to explore it while enjoying gaming. To achieve this goal, game developers design widgets in different shapes and colors. For example, after we investigate GUIs in a game *Elysium of Legend* and GUIs in a hotel booking application of RICO dataset. We collect the number of heterogeneous widgets and regular widgets; the data is shown in Figure 3.8. In this booking application, there are 14 GUI elements in total and only 4 heterogeneous GUIs (e.g., sharing widget, map widget). In applications, there are more than 60 types of widgets, and 23 of them are heterogeneous widgets. These irregular widgets are often in various shapes (ancient buildings, cat's claws), as shown in Figure 3.8. Moreover, these widgets are often placed on background pictures, making it difficult for models to separate background textures from widgets. In our experiments, all models missed the three irregular widgets.

Art style difference also causes huge diversity between games. We investigate the return widgets in nine games in our dataset, and results are shown in Figure 3.9. We surprisingly find that there exist no two similar return widgets. Some return widgets are even reshaped in uncommon style (e.g., the returned widget in the left part of the second row), which confuses models to identify them properly. The art style gap between games is often large, not to mention

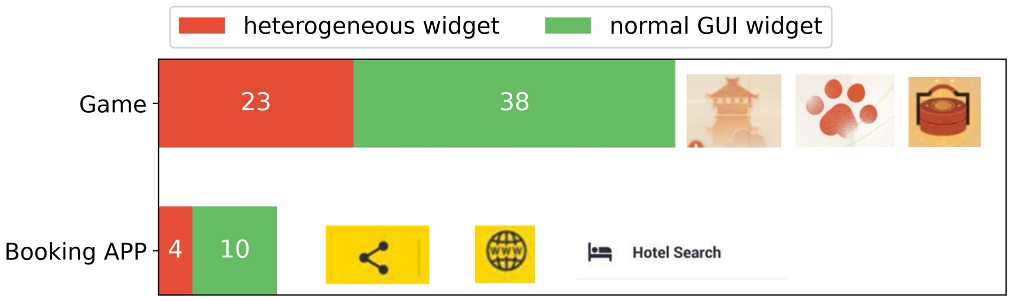


Figure 3.8: Comparing the number of GUIs categories in a game and an Android application. Followed by examples of heterogeneous GUI elements in this game.

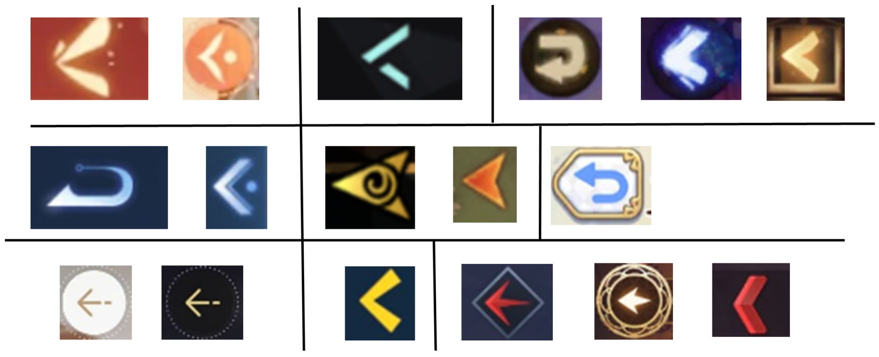


Figure 3.9: Return widgets in different games. Widgets in different games are separated by solid lines.

the gap between regular applications and games. This also explains the reason why the performance of models trained on the RICO dataset but tested on the game dataset drops sharply.

3.4.5 Summary of Findings

3.4.5.1 Answers to the Research Questions

Through the analysis of the experiments above, we found that:

- **Answer to RQ2:** The models (i.e., Faster RCNN and YOLO) cannot generalize well across the game and the RICO dataset. The experimental results show that the performance of models drops sharply on the game dataset. Meanwhile, models trained on game datasets are not accurate for detecting regular GUI widgets in non-game applications.

- **Answer to RQ3:** The models achieve unsatisfactory performance because (1) widgets are compactly placed in game GUIs, making the density of GUI widgets in the game far larger than that in regular applications, (2) heterogeneous GUI shapes and high diversity of GUI styles between games make it difficult for models to detect widget regions.

3.4.5.2 Research Direction Highlight

Based on our findings, we provide insightful suggestions for future research directions.

Enhancing Dataset. In our study, we automatically and manually labeled 2,993 screenshots of 9 games. Compared with previous works, RICO dataset includes more than 60K screenshots. Additionally, in order to assist better game GUI testing in industries, the dataset only provides coordinates of clickable widgets. This can be further enhanced by adding support for multiple types of widgets (e.g., text widgets, picture widgets) for facilitating future research on game GUIs. Additionally, the dataset can be enhanced by collecting various types of games (e.g., action games, adventure games, first-person shooting games).

Game Style Comparison. Recall that the model performs badly across different datasets. In fact, some mobile games share similar art styles with particular games. For example, the art style of the game Arena of Valor is similar to the game Onmyoji Arena (not included in our dataset in this paper but also famous on Google Play Store). If we find a metric to evaluate the similarity between the styles, we can summarize a minimum set of games that covers most game styles. In further training, models could be trained on this minimum set to save a considerable amount of time.

Applying GUI Detection In Industry. As we have discussed with testing developers from industries about applying widget detection methods in industry game testing, the developers agreed that GUI widget detection techniques are important for game designs and game testing. Further, a game GUI dataset detection technique can help a freshman engineer rapidly become familiar with developments in new games. Currently, we are engaged in the implementation of this service and applying it in industry game testing in the near future.

3.4.6 Threats to Validity

We note that randomness is an inevitable factor when applying deep learning models. To alleviate the effect of this factor, we repeat the experiments mentioned in our study 5 times and record the average values. The selection of games could be biased. In our study, we adopted 6 games released by them. To counteract the bias, we adopt additional games released by other companies (e.g., *Tencent Games, Cygames*). Additionally, in previous works, other models (e.g., CenterNet [96], EAST [97], Grad-CAM [98]) are adopted in their methods. We choose to select Faster RCNN and YOLO in our experiments because they are commonly adopted in most similar studies. On the other hand, the recall rate, which is adopted as our metric, maybe a potential threat. Generally, the recall performance of a model is difficult to evaluate due to the lack of ground truth. In our study, the labels of clickable widgets are mostly processed by us and are prone to introducing incompleteness. To alleviate this, we make our best attempt to check the labels 3 times. On this basis, despite our labels being unable to be 100% complete, our dataset however provides a convincing benchmark for evaluating model performance.

3.5 Chapter Conclusion

In this study, we first conduct a survey in mobile game testing, including scrum interviews and questionnaires. The survey results show that applying an object detection method to detect game GUI widgets can be the pillar to boost game testing efficiency in practice. To this end, we develop a method to automatically collect the GUI of industrial games and a method for data-cleaning. The evaluations show that (1) existing general-purpose GUI methods cannot perform well on games and (2) the unsatisfactory performance of existing methods is mainly caused by the compactly placed GUI widgets and the diverse GUI shapes.

CHAPTER

FOUR

STUDY ON VULNERABILITIES AND EXISTING TOOLS IN SMART CONTRACT

4.1 Introduction

Powered by the Blockchain technique [99], smart contracts [100] have attracted much attention and have been applied in various industries, e.g., financial services, supply chains, smart traffic, and IoTs. Solidity is the most popular language for smart contracts for its mature tool support and simplicity. However, the public has witnessed several severe security incidents, including the notorious DAO attack [101] and the Parity wallet hack [102]. According to previous reports [23, 103], up to 16 types of security vulnerabilities were found in Solidity programs. These security issues undermine the confidence of people who have executed transactions via smart contracts and eventually affect the trust in the Blockchain ecosystem.

Witnessing the severity and urgency of this problem, researchers and security practitioners have endeavored to develop automated security scanners [25, 26, 28, 54, 57]. Existing state-of-the-art scanners usually adopt rule-based methods for vulnerability detection. Slither [25] supports 39 hard-coded static rules; Securify [28] supports 15 rules for verifying the extracted path constraints from the contract with the SMT solvers [104]; Oyente [26] supports eight rules for generating assertions for verifying the vulnerabilities. Each rule represents a pattern of vulnerable contracts, which warns the programmers to avoid potential risks before deploying the contracts.

Although experiments have demonstrated their effectiveness, it is notable that human experts manually craft the rules behind these scanners. *The manually predefined rules can be obsolete*, because 1) previously unseen vulnerable code may be introduced, which cannot be captured by the hard-coded rules, and 2) new defense mechanisms (i.e., programming skills to prevent bugs) may have successfully mitigated the vulnerabilities. However, the code may still match the predefined vulnerable pattern or rules. Therefore, most updated rules should be learned to distinguish vulnerable contracts from robust ones.

In this work, we alleviate the incompleteness of detection rules by combining vulnerability signatures abstracted from both vulnerable and benign contracts (i.e., vulnerable signature and benign signature). The vulnerable signature is designed to match the commonalities of a particular vulnerability. Comparatively, the benign signature is abstracted from falsely reported contracts to reduce false alarms. For each vulnerability, we adopt vulnerable and benign signatures to synthesize the detection rules of Vulpedia. Note that Vulpedia is built upon the relaxed assumption that the contract's owner is not malicious. Detecting malicious contracts (e.g., contract with backdoors, exploit code) is different from vulnerability detection (i.e., the target of Vulpedia). Based on this assumption, the operations related to the contract owner are all deemed as vulnerability defense behaviors. Compared with previous work, the synthesized rules are more updated and expressive than the predefined rules in the state-of-the-art vulnerability scanners, capturing a lot of unseen patterns in practice.

In our implementation, we first collect truly and falsely reported contracts by applying three state-of-the-art vulnerability scanners (i.e., Slither [25], Oyente [26], and Securify [28]) and manually evaluate their correctness. Based on the results, analyzing truly reported vulnerable contracts allows us to capture salient program signatures responsible for vulnerable contracts. In contrast, analyzing falsely reported vulnerable contracts will enable us to capture noticeable signatures to avoid false alarms. Next, we categorize the contracts by their vulnerability types (e.g., Reentrancy, Unchecked Low-level-call, etc.) and alarm types (i.e., true or false). For each category, we cluster the contracts based on their tree edit distance [105] and then extract program feature commonalities from each cluster's PDGs (program dependency graph) to summarize vulnerability signatures. Finally, we abstract four vulnerable signatures and six benign signatures. They are integrated as four detection rules regarding four vulnerabilities (i.e., Reentrancy, SelfDestruct, Tx-origin, and Unexpected-Revert).

We conduct our signature abstraction on a set of 76,354 intelligent contracts and evaluations on a set of 17,770 contracts, respectively. The evaluation results show that, compared with the state-of-the-art vulnerability scanners (i.e., Slither, Oyente, Smartcheck, and Securify), our approach achieves outstanding accuracy on four vulnerabilities and leading recall on three vulnerabilities.

To summarize, we make the following contributions:

1. We propose an approach to abstract vulnerability signatures and compose detection rules to report the vulnerability. The learned rules are more expressive than rules of the state-of-the-art scanners, reporting vulnerabilities with better completeness and soundness
2. On the 17,770 contracts crawled from Google, Vulpedia yields the best precision on four vulnerabilities and leading recall on three ones, in comparison with the other state-of-the-art scanners.
3. Experiments show that Vulpedia is efficient in vulnerability detection. The detection speed of Vulpedia on 17,770 contracts is far faster than Oyente and Securify.

In this chapter, we organize our content like this: In the Background, we first introduce the different types of vulnerabilities we address in our study and explain why the state-of-the-art tools fail. In the Overview, we illustrate the basic steps of our proposed tool, namely Vulpedia. In the Empirical Study we conduct an empirical study and introduce our method of signature abstraction. We also elaborate on the effectiveness of signatures with examples. In experiments, we compare Vulpedia with the other state-of-arts using 17,770 real-world contracts deployed on Ethereum. The related work briefly introduces the related work, and the conclusion summarizes this study.

4.2 Overview

Figure 4.1 shows the workflow of abstracting vulnerability signatures for Vulpedia. The workflow can be roughly grouped into four steps: 1) The pre-detection of existing tools; 2) Vulnerability report inspection; 3) AST clustering and signature abstraction; 4) Rule composition. Note that manual efforts are involved in steps two and step 4.

In the first two steps, we systematically evaluate (1) how accurately state-of-the-art tools can report vulnerable smart contracts and (2) under what condition those tools can be ineffective. We collect the reports of the state-of-the-art tools on a training dataset of 76,354 contracts. Then, we employ three experienced smart contract developers to manually confirm the reports of the tools and categorize them into two groups: truly alarmed vulnerable contracts and falsely alarmed vulnerable contracts.

In the last two steps, we first calculate the tree edit distance based on the ASTs of contracts in a particular vulnerability type and cluster the contracts of the type by defining the contract similarity. Next, we abstract common nodes from the PDGs (program dependency graph) of each cluster to summarize signatures (e.g., as shown in Figure 4.2). From truly vulnerable contracts, we summarize vulnerable signatures. In contrast, from falsely alarmed vulnerable contracts, we summarize benign signatures. Finally, we manually integrate vulnerable and benign signatures into vulnerability detection rules.

After we equip our Vulpedia detector with the composed rules, the detector takes unknown contracts as inputs and generates vulnerability reports based on the signatures. Specifically, the detector first conduct preprocessing on the input smart contract code. The detector extracts normalized AST from the contract. Based on this normalized AST, the detector conducts a PDG extraction. Meanwhile, the detector extracts existing signatures from the vulnerability signature database. Lastly, the detector produces detection reports based on the comparison results. If the PDG matches vulnerable signatures

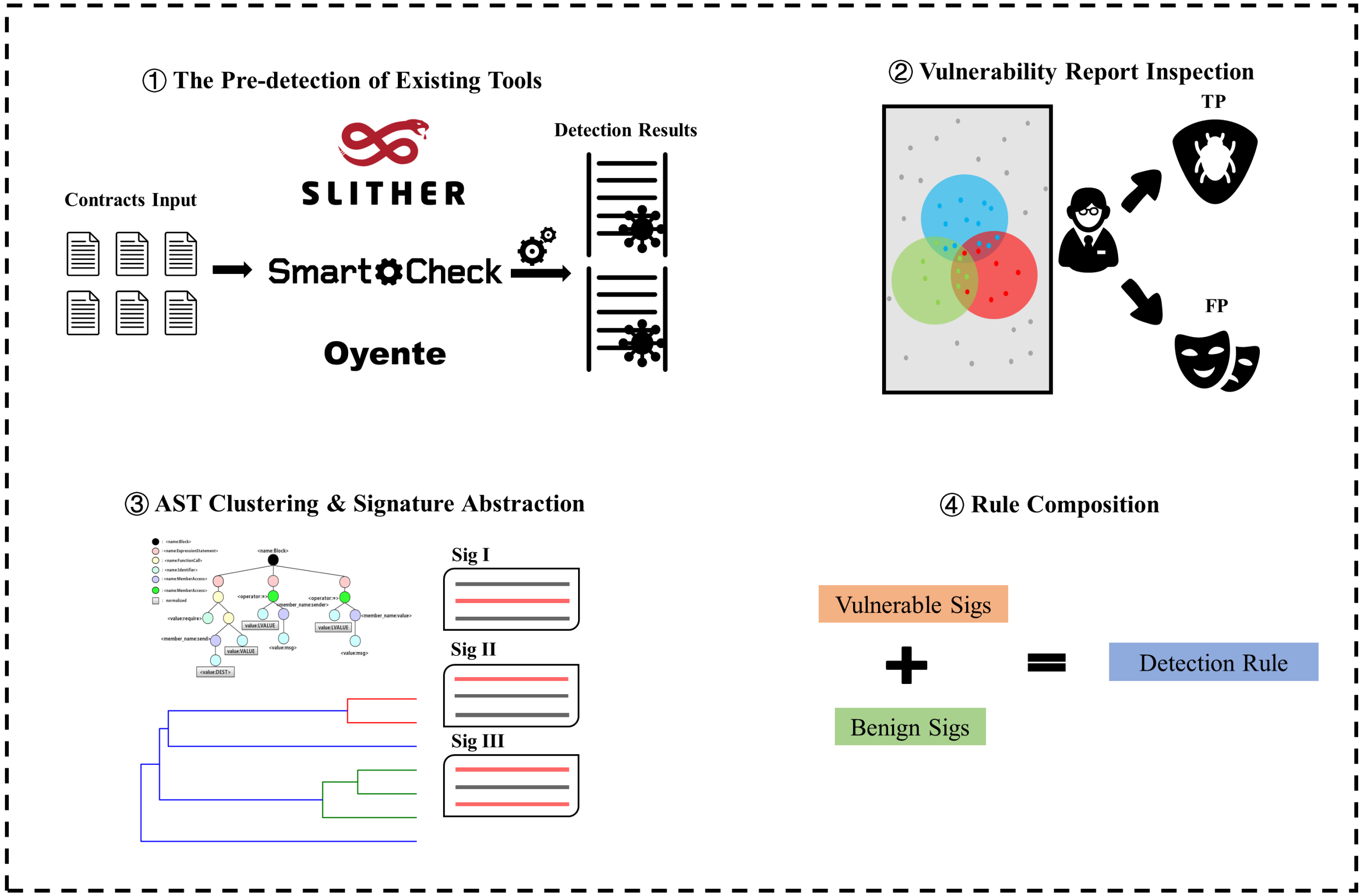


Figure 4.1: The workflow of extracting vulnerability signatures of Vulpedia.

but is not matched with benign signatures, the contract will be deemed vulnerable; otherwise, the detector will produce a non-vulnerable report.

4.3 Empirical Study of Signature Abstraction

In this section, we first illustrate how we empirically collect contracts in this study. We report how we select the vulnerability scanners and how we construct a contract dataset. Next, we introduce our method of 1) clustering similar contracts by comparing tree edit distance, 2) abstracting commonalities from PDGs of clusters as signatures, and 3) detection rules composition based on the abstracted signatures. Finally, we elaborate on the signatures with examples to evidence their representativeness of them.

4.3.1 Selected Scanners and Dataset

Table 4.1: The state-of-art tools for Solidity analysis.

Tool Name	Method	Technique	Open Source	Implementation
Mythril [64]	Dynamic	Constraint Solving	●	Python
MythX [65]	Dynamic	Constraint Solving	❍	N.A.
Slither [25]	Static	CFG Analysis	●	Python
Echidna [66]	Dynamic	Fuzzy Testing	●	Haskell
Manticore [63]	Dynamic	Testing	●	Python
Oyente [26]	Dynamic	Constraint Solving	●	Python
Securify	Static	Datalog Analysis	●	Java
Smartcheck [54]	Static	AST Analysis	●	Java
Octopus [60]	Static	Reverse Analysis	●	Python
Zeus [57]	Static	Formal Verification	❍	N.A.
ContractFuzzer [32]	Dynamic	Fuzzy Testing	●	Go

4.3.1.1 Choice of Scanners and Vulnerability Types

Overall, we select vulnerability scanners based on how practical they can be used in real-world scenarios. We investigate a list of static analyzers, including

Slither [25], Oyente [26], Zeus [57], Smartcheck [54], and MythX [65]. These tools utilize manually defined detection rules to detect vulnerabilities. The rules could match vulnerabilities in some cases but also generate many false reports. We also investigate dynamic detectors like Mythril [64], Contract-Fuzzer [32], Echidna [66], and Manticore [63]. They exercise programs and check the runtime status of functions to find vulnerabilities. These analyzers often achieve high detection precision but suffer from limited scalability. Additionally, we investigate other analyzing tools (e.g., Solidity reverse engineering tool Octopus [60]) to facilitate our exploiting contracts. A summary of the above tools can be found at Table 4.1. In our study, some tools are not selected because they are not open-sourced (Zeus [57], MythX [65]), not related to our task (Echidna [66], Octopus [60]) and efficiency concerns (Mythril [64], ContractFuzzer [32], Manticore [63]).

Finally, we choose Slither v.0.4.0, Oyente v0.2.7, and Smartcheck v2.0 as our scanners.

4.3.1.2 Dataset for Empirical Study

We implement a web crawler to download Solidity files from accounts of Etherscan [106], a famous third-party website on Ethereum Block Explorer. Etherscan provides APIs for downloading transaction information (e.g., transaction addresses, time). Our crawler can be accessed at `https://github.com/ToolmanInside/smart_contract_crawler`. The crawler adopts a random search strategy on the website of Etherscan to ensure the downloaded contracts are randomly sampled.

We choose contracts deployed by Solidity 0.4.25 and 0.4.24. The reasons are two folds: 1) as reported in [107], 54.27% Solidity smart contracts are in 0.4 version, and the 0.4.24 and 0.4.25 are the latest versions in Solidity 0.4; 2) the versions 0.4.24 and 0.4.25 are supported by most analyzers so that they facilitate our study. Additionally, we find that the downloaded dataset has redundant contracts (contracts that share commonality with others). Regarding these redundant contracts, we remove contracts that are exactly the same as others and contracts that are only different in transfer address with others. Finally, we got 76,354 contracts for our dataset.

Table 4.2 shows the number of contracts we collected in this study. Overall, among 76,354 contracts, the three tools report 508 true vulnerable contracts, albeit 3,496 false vulnerable ones. Table 4.3 shows the details on the number of reported contracts and precision performance of each tool. We observed that all the tools have a large number of false alarms. This is due to contract programmers having invented many heuristics to detect potential vulnerabilities. In other words, most existing detection rules are obsolete. It motivates us to pursue (and generate) a more expressive and fine-grained rule to mitigate false alarms.

Table 4.2: Number of collected contracts for each category

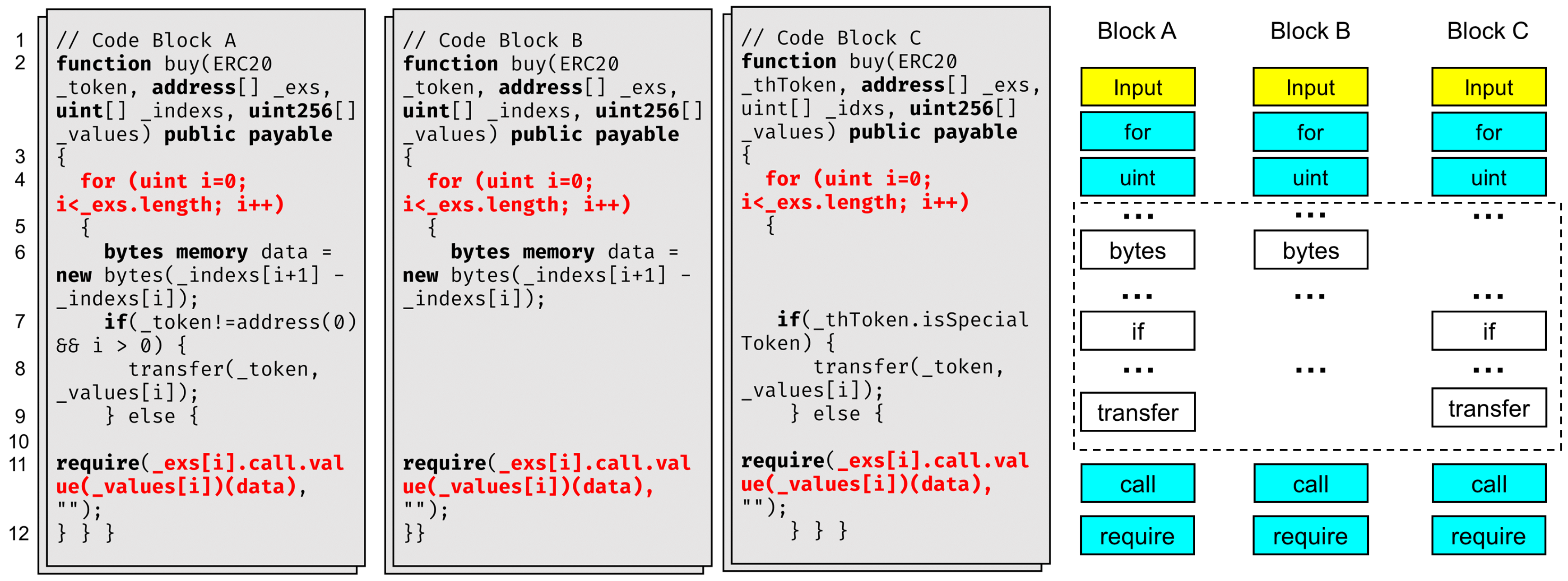
Alarm Type	**RE**	**TX**	**UR**	**SD**
True Positive	46	38	421	3
False Positive	720	179	2,546	51

Table 4.3: The precision performance of three tools Slither, Oyente, and Smartcheck on four vulnerabilities.

Vulnerability	Slither	Oyente	Smartcheck
Reentrancy	623 (3.53%)	143 (16.78%)	N.A.
Abused Tx.origin	67 (28.35%)	N.A.	150 (12.66%)
Unexpected Revert	2,678 (8.25%)	N.A.	289 (69.20%)
Self Destruct	54 (5.56%)	N.A.	N.A.

4.3.2 Vulnerability Rule Abstraction

In this section, we introduce the definition of signature and show how we cluster and abstract the vulnerable/benign signatures from each cluster.



```
// Code Block A
function buy(ERC20
_token, address[] _exs,
uint[] _indexs, uint256[]
_values) public payable
{
  for (uint i=0;
i<_exs.length; i++)
  {
    bytes memory data =
new bytes(_indexs[i+1] -
_indexs[i]);
    if(_token!=address(0)
&& i > 0) {
      transfer(_token,
_values[i]);
    } else {

require(_exs[i].call.val
ue(_values[i])(data),
"");
} } }
```

```
// Code Block B
function buy(ERC20
_token, address[] _exs,
uint[] _indexs, uint256[]
_values) public payable
{
  for (uint i=0;
i<_exs.length; i++)
  {
    bytes memory data =
new bytes(_indexs[i+1] -
_indexs[i]);

require(_exs[i].call.val
ue(_values[i])(data),
"");
}}
```

```
// Code Block C
function buy(ERC20
_thToken, address[] _exs,
uint[] _idxs, uint256[]
_values) public payable
{
  for (uint i=0;
i<_exs.length; i++)
  {

    if(_thToken.isSpecial
Token) {
      transfer(_token,
_values[i]);
    } else {

require(_exs[i].call.val
ue(_values[i])(data),
"");
    } } }
```

Figure 4.2: Three similar code blocks of Unexpected Revert that are found in real-world contracts. Based on their tree edit distance, we cluster them together and abstract a graph skeleton from their PDG. The yellow boxes denote function inputs, the blue boxes denote common nodes on PDG, and the white boxes in the dotted box represent different nodes.

4.3.2.1 Definition

We define a vulnerability rule for a Solidity contract as following BNF:

$\langle rule \rangle$ ::= $\langle comp\_sig \rangle$

$\langle comp\_sig \rangle$ ::= $\neg\langle comp\_sig \rangle$ | $(\langle comp\_sig \rangle \vee \langle comp\_sig \rangle)$ | $(\langle comp\_sig \rangle \wedge \langle comp\_sig \rangle)$ |
$(\langle comp\_sig \rangle \succ \langle comp\_sig \rangle)$ | $\langle sig \rangle$

$\langle sig \rangle$ ::= *DataDep*(*X*,*Y*) | *ControlDep*(*X*,*Y*) | *ForLoop* |
IsInstance(*X*,*Y*) | *Call*(*L*,*X*) | *SelfDestruct*(X) | msg.sender | tx.origin |

Here, the detection rule is a composite of signatures. A composite signature is a negation ($\neg$) of itself, or conjunction ($\wedge$), union ($\vee$), succeed ($\succ$) with another composite signature. A composite signature can also be a single vulnerability signature. Specifically, the vulnerability signature indicates basic program relationships and built-in keywords of Solidity language. For example, the data dependency *DataDep*(X,Y) relationship denotes that variable X has data dependency to Y (i.e., variable assignment operations). The control dependency *ControlDep(X,Y)* indicates assertion operations (e.g., require, assert) between variables X and Y. The for loop *ForLoop* denotes the function body has a for loop statement. The *IsInstance(X,Y)* denotes the variable X is a type of variable Y. Call operation *CALL(L, X)* includes low-level calls (e.g., `call.value()` and `send()` in Solidity) and high-level calls (i.e., user-defined function calls). Here, variable L represents the result of call operations, and variable X represents the parameters required by the call. *SelfDesutrct*(X) is a built-in function call in Solidity. Once it is called, the service of the current contract is stopped, and the rest balance is transferred to an arbitrary receiver X. The msg.sender and tx.origin are built-in variables. Specifically, msg.sender denotes the address of the current contract, and tx.origin denotes the origin of call chains [22].

4.3.2.2 Contract Clustering

In this section, we first define contract similarity on normalized ASTs, and then we cluster similar trees by using a hierarchical clustering algorithm. The clustering procedure can be found in lines 1 to line 10 in the Algorithm 3.

Algorithm 3: Contract Clustering and Signature Abstraction Algorithm

```
input  : SourceCode, source code of smart contracts
output: SignatureCands, abstracted signature candidates
1  // Contract Clustering Process
2  ASTs = $getAST(SourceCode)$
3  nASTs = $ASTNomalization(ASTs)$
4  distanceMatrix = $List[N, N]$
5  // N is the number of trees
6  foreach $idx\ i \in range(nASTs)$ do
7      foreach $idx\ j \in range(nASTs) and i \neq j$ do
8          treeEdtDist = ARTED(nASTs[i], nASTs[j])
9          // Calculate the distance between two trees
10         distanceMatrix[i, j] = treeEdtDist
11 Clusters = $hierarchicalClustering(distanceMatrix)$
12 // Signature Abstraction Process
13 $SignatureCands \leftarrow \emptyset$
14 foreach $cluster\ c \in Clusters$ do
15     $PDGs \leftarrow \emptyset$
16     foreach $tree\ t \in c$ do
17         PDG $p \leftarrow getPDG(t)$
18         $pn \leftarrow PDGNormalization(p)$
19         $PDGs \leftarrow PDGs \cup pn$
20     $commonSeq \leftarrow LCS(PDGs)$
21     $SignatureCands \leftarrow SignatureCands \cup commonSeq$
22 return $SignatureCands$
```

Contract Similarity. We define the contract similarity by considering both the semantic and structural information of the code. To this end, we use AST (Abstract Syntax Tree) to represent the code of the functions of each contract. For each AST of a Solidity function, we normalize the concrete nodes in the AST for retaining core information and abstracting away unimportant details such as variable names or constant values, as shown in line 2 of Algorithm 3. For each AST corresponding to a function, we just retain the information such as node type, name, parameter, and return value (if contained). For the variable names (e.g., `_indexs` in code block A and code block B of Figure 4.2), they will be normalized with the token asterisk "$*$". Similarly, we repeat the same normalization for constant values of the types `string`, `int`, `bytes`, or `uint`.

Given two trees, we use the tree edit distance between two normalized ASTs as their distance. The AST normalization process is shown in line 3 of Algorithm 3. In this work, we apply a robust algorithm for the tree edit distance (ARTED) [105], which computes the optimal path strategy by performing an exhaustive search in the space of all possible path strategies. Here, *path strategy* refers to a mapping between two paths of the two input trees (or subtrees), as the distance between two (sub)trees is the minimum distance of four smaller problems, i.e., (1) the edit distance between two empty trees, (2) the edit distance of transferring a tree F to an empty tree, (3) the edit distance of transferring an empty tree to a tree F and (4) the edit distance of transferring a tree F to another tree G. Note that though ARTED runs in *quadratic* time and space complexity, it is guaranteed to perform as good or better than its competitors [105].

Contract Clustering. We cluster the ASTs via hierarchical clustering algorithm with complete linkage [108], as shown in line 4 to line 10 in Algorithm 3. Then, we group the codes in Figure 4.2 with considerable modification. We deem that the ASTs in each cluster share commonalities as a feature (or signature) for a vulnerability category.

4.3.2.3 Signature Abstraction

After clustering contract functions with AST, we abstract signatures by referring to their PDG (Program Dependency Graph) information. The reason lies in that PDG allows us to capture the code semantic features like control and data dependencies.

PDG Representation. For each AST, we transfer its code into a PDG, including all its dependent code elements such as global variables and called functions, as shown in line 13 to line 17 in Algorithm 3. In a PDG, each of its nodes is an instruction, and the edge between nodes indicates data dependency, control dependency, and call relation between the nodes. Thus, given a cluster containing N Solidity functions, we reduce it into a problem of finding the common subgraph of N PDGs. The normalization of PDGs is shown in line 18 in Algorithm 3.

PDG Matching. The graph matching problem is an NP-complete problem. We simplify the problem with the following steps. Before matching, we also abstract away variable names and constant values in the PDGs as we do that for AST. Next, we simplify the calculation by flattening the graph into a node sequence (via depth-first order search) and aligning the sequences by LCS algorithm [109], as shown in line 20 in Algorithm 3. The aligned graph nodes are considered commonalities shared by the code in the same cluster.

As a result, the signature abstracted from a cluster is essentially a graph skeleton, as shown in Figure 4.2. Then, we manually inspect those skeletons and refine them into usable signatures. The refining process requires manual efforts because some signatures are semantically similar to others but different in syntax. These signatures require to be filtered out by human experts. After we repeat the above procedures on both vulnerable and benign contracts, we construct a set of vulnerable and benign signatures.

4.3.2.4 Rule Composition

In this study, we follow the following heuristics to incorporate the signatures into a rule. Generally, a *detection rule is a composite boolean expression of vulnerability signatures*. Given a vulnerability category, a detection rule first requires the input contract to match with the vulnerable signatures. The vulnerable signatures are essential ingredients for forming a vulnerability. Therefore, if the input contract is not matched with vulnerable signatures, the contract should be considered invulnerable. Next, the input contract is required not to match with benign signatures. The benign signatures are the best practices

Table 4.4: Extracted Signatures from Different Vulnerability Categories

ID	Vulnerability	V/B	Signature
1	Reentrancy	V	$DataDep(\_,X) \succ Call(\_,X)$
2		B	$ControlDep(\text{msg.sender},X) \succ DataDep(\_,X) \succ Call(\_,X)$
3		B	$DataDep(\_,X) \succ IsInstance(X,\text{addr}) \succ Call(\_,X)$
4		B	$ControlDep(Y,\_) \succ DataDep(\_,X) \succ Call(\_,X) \succ DataDep(Y,\_)$
5	Unexpected Revert	V	$ForLoop \succ Call(L,X) \succ ControlDep(L,\_)$
6		B	$ForLoop \succ \big(IsInstance(X,\text{addr}) \wedge Call(L,X)\,\big) \succ ControlDep(L,\_)$
7	Abuse of Tx.origin	V	$DataDep(\text{tx.origin},X) \succ ControlDep(X,\_)$
8		B	$DataDep(\text{msg.sender},Y) \succ DataDep(\text{tx.origin},X) \succ ControlDep(X,Y)$
9	SelfDestruct	V	$DataDep(\_,X) \succ SelfDestruct(\text{X})$
10		B	$ControlDep(msg.sender,X) \succ DataDep(\_,X) \succ SelfDestruct(\text{X})$

to defend against vulnerabilities. If the input contract matches with them, it suggests that the contract is capable of defending vulnerabilities and should not be reported as vulnerable.

Table 4.5: Detection rules for each vulnerability

ID	Vulnerability	Rule
1	Reentrancy	SIG1 $\wedge \neg$ (SIG2 $\vee$ SIG3 $\vee$ SIG4)
2	Revert	SIG5 $\wedge \neg$ SIG6
3	Tx.origin	SIG7 $\wedge \neg$ SIG8
4	Self-destruct	SIG9 $\wedge \neg$ SIG10

4.3.3 Case Study: Abstracted Signatures

We applied the three chosen scanners to 76,354 contracts. Overall, Slither reports the most vulnerabilities, in total, 3,422 (623 + 67 + 2,678 + 54) candidates covering four types. In contrast, Smartcheck reports 439 (150 + 289) candidates, and Oyente reports only 143 candidates. After they are processed by our methods, we abstract 4 vulnerable signatures and 6 benign signatures, as shown in Table 4.4. Based on these signatures, we further integrate them into

4 detection rules, as shown in Table 4.5. In this section, we elaborate on the signatures with examples to evidence their representativeness.

Signature of Reentrancy. We extract 4 signatures from TPs and FPs of reported reentrancy vulnerabilities, including 1 vulnerable signature (**SIG1**) and 3 benign signatures (**SIG2**, **SIG3**, **SIG4**).

SIG1 is abstracted from general patterns of reentrancy vulnerabilities. This signature consists of two parts: (1) the read or write operation of variable X (i.e., *DataDep*(_,X)) and (2) the call operation with the parameter variable X (i.e., *Call*(_,X)).

SIG2 adds various forms of checks (i.e., in `require` or `assert`) for msg.sender compared with SIG1. For example, SIG2 checks whether the identity of msg.sender satisfies certain conditions (e.g., equal to the owner, with a good reputation, or having the dealing history) before calling the external payment functions. With the identity check, the function is only accessible to related users, blocking the malicious attack from attackers. An example of this signature can be found at Code 1.

SIG3 describes a falsely reported case of transferring the balance to a fixed address. In Code 5, the function `closePosition` sends the balance to a token `bancorToken`, which is assigned with a fixed address at line 2. According to the detection rule of Slither (See Equation 2.1), this code is a vulnerability because — (1) it reads the `public` variable `agets[_idx]`; (2) then calls external function `bancorToken.transfer()`; (3) last, writes to the `public` variable `agets[_idx]`. However, in practice, this contract can never be easily exploited to steal ethers due to the hard-coded address constant (i.e., `0x1F...FF1C`). Note that the constant address can be a malicious address. Under such circumstances, this address cannot protect the contract. However, this case is very rare. Therefore, we choose to trust the creator of the contract as well as the designated addresses are benign.

SIG4 is to prevent the recursive entrance of the function — eliminating the issue from the root. For instance, in Code 6, the internal instance variable `reEntered` will be checked at line 5 before processing the business logic between lines 8 and 10. To prevent the reentering due to calling `buyAndSet` function at line 9, `reEntered` will be switched to `true`; after the transaction is done, it will be reverted to `false` to allow other transactions.

Code 5 A real case of using SIG3 (a hard-coded address at line 3), an FP of *reentrancy* for Slither.

```
1  contract BancorLender {
2      ERC constant public bancorToken =
3        ERC(0x1f573d6fb3f13d689ff844b4ce37794d79a7ff1c);
4      function closePosition(uint _idx) public {
5        ...
6        bancorToken.transfer(agets[_idx].lender, amount);
7        return;
8        } }
```

Code 6 A real case of using SIG4 (an execution lock of `reEntered`), an FP DataDepof *reentrancy* for Slither.

```
contract ZethrBankroll is ERC223Receiving {
    ZTHInterface public ZTHTKN;
    bool internal reEntered;
    function receiveDividends() public payable {
        if (!reEntered) {
        ...
        if (ActualBalance > 0.01 ether) {
            reEntered = true;
            ZTHTKN.buyAndSet.value(ActualBalance)(address(0x0), 33, "");
        }
    } } }
```

Code 7 A real FP of Unexpected Revert reported by Smartcheck, where only one account is involved (SIG6).

```
function withdraw() private {
    for(uint i = 0; i < player_[uid].planCount; i++) {
        ...
        address sender = msg.sender;
        sender.transfer(amount);
    } }
```

Signature of Unexpected Revert. We extract 2 signatures from reported Unexpected Revert vulnerabilities, including 1 vulnerable signature **SIG5** and 1 benign signature **SIG6**.

SIG5 represents general patterns of Unexpected Revert vulnerabilities. This signature consists of three parts: (1) the for loop program structure (i.e., *ForLoop*); (2) the call operation of the variable X (i.e., *Call*(_,X)); (3) the result of call operation is further checked by assertions.

According to the recent technical article [110], the rules of *Call/Transaction in Loop* are neither sound nor complete to cover most of the unexpected revert cases. At least, modifier `require` is often ignored, which makes Slither and Smartcheck incapable to check possible revert operations on multiple account addresses. Here, multiple accounts must be involved to exploit this attack — the failure on one account blocks other accounts via reverting the operations for the whole loop. Hence, in the example of Code 7, the operations in the loop are all on the same account (i.e., `sender` at line 5), and potential revert will not affect other accounts. Therefore, the transfer operation of which the target is a single address is considered as **SIG6**.

Signatures of `Tx.Origin` Abusing. We extract 2 signatures from the truly vulnerable contracts and falsely reported contracts, including 1 vulnerable signature (**SIG7**) and 3 benign signatures (**SIG8**).

For **SIG7**, this signature is extracted from general patterns of tx.origin vulnerabilities. This vulnerability first reads the value of tx.origin, followed by an assignment to variable X (i.e., *DataDep*(tx.origin,X)). After this, the function has an assertion to this variable (i.e., *ControlDep*(X,_)). While we extract signatures from the TPs of vulnerabilities, we find that our **SIG7** is slightly looser than the detection rule in Slither. Slither skips the function if there exists a read operation to a particular variable msg.sender, ignoring that some of these variables are irrelevant to tx.origin. In order not to overlook potential vulnerabilities, our **SIG7** only requires a read of tx.origin, followed by an assertion on this variable.

For **SIG8**, we observe that Smartcheck reports much more cases (210) than Slither (34) but has a lower precision performance than Slither. After our investigation, we find that the incorrect reports of Smartcheck are due to the unsound rules (as shown in Equation 4.1). That is, Smartcheck simply reports vulnerability once tx.origin appears in assertion statements. However, under some circumstances (e.g., comparing msg.sender with tx.origin), the use of tx.origin should not be reported. We summarize the **SIG8** based on the FPs of Smartcheck.

$$\boxed{\mathit{DataDep}(\mathit{tx.Origin}, X) \succ \mathit{ControlDep}(X, \_) \Rightarrow \text{Tx.Origin abusing}} \tag{4.1}$$

Code 8 A real FP of *self-destruct abusing* by Slither, as `selfdestruct()` is used under two checks at line 2,3 (SIG10).

```
1  function destroyDeed() public {
2      require(msg.sender == owner);
3      if (owner.send(address(this).balance)) {
4          selfdestruct(burn);}
5  }
```

Signature of Self-destruct Abusing. We extract 2 signatures from the self-destruct vulnerabilities, including 1 vulnerable signature **SIG9** and 1 benign signature **SIG10**.

SIG9 is extracted from general patterns of self-destruct vulnerabilities. This signature consists of two parts: (1) the read or write operation of variable X (i.e., $\mathit{DataDep}(\_,X)$) and (2) the call operation of the self-destruct with the parameter X (i.e., $\mathit{SelfDestruct}(X)$).

For **SIG10**, we extract this signature from the FPs of tools. In the existing scanners, only Slither detects the misuse of self-destruct, which is called suicidal detection. In total, Slither reports 54 cases of suicidal via its built-in rule — as long as a function *SelfDestruct* is used, no matter what the context is, Slither will report it. Obviously, the rule of Slither is too simple and too general. It mainly works for directly calling *SelfDestruct* without permission control or conditions of business logic — under such circumstances (3 out of 54), the Slither rule can help to detect the abuse. In practice, in most cases (51 out of 54), *SelfDestruct* is called with the `admin` or `owner` permission control or under some strict conditions in business logic. For example, *SelfDestruct* is indeed required in the business logic at line 2 of Code 8. As the owner wants to stop the service of the contract via calling *SelfDestruct*, after the transactions are all done, the contract becomes inactive. Note that parameter `burn` is

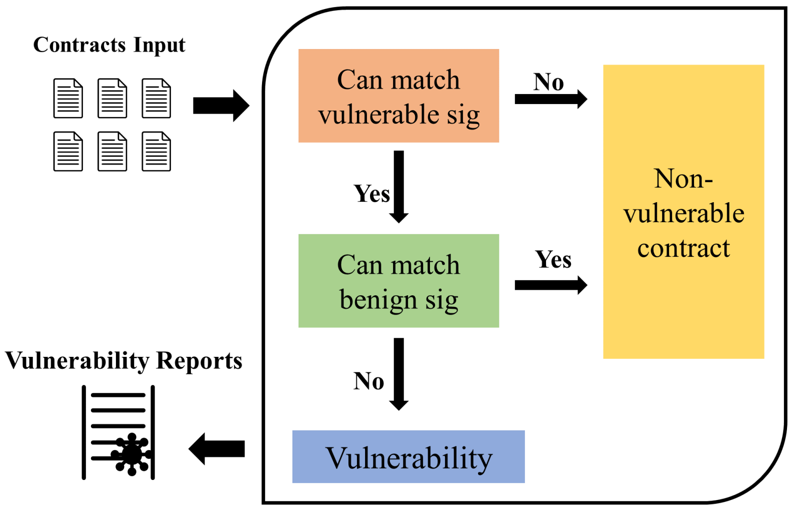


Figure 4.3: The detection workflow of Vulpedia.

just padded to call *SelfDestruct* in the correct way. Hence, we summarize the **SIG10**, adding a strict condition control or a self-defined modifier for identity check when using *SelfDestruct*.

In brief, for a vulnerability type, we use vulnerable signatures to match potential vulnerabilities, which yields a better recall. Then, we leverage corresponding benign signatures to filter out false reports.

4.3.4 Vulnerability Detection

The implementation of the vulnerability detection of Vulpedia is based on the previously abstracted signatures and integrated detection rules but is slightly different from them. The workflow of detection is shown in Figure 4.3. Specifically, in this workflow, Vulpedia reports vulnerability only when the vulnerable signatures are matched meanwhile, the benign signatures are not matched. That is, the vulnerable and benign signatures are separate things. However, in the previous subsection, the signatures are combined to form detection rules. The reason is that our benign signatures are designed to filter out false positive reports. All detection rules shown in Table 4.5 follow the pattern that the vulnerable signatures should be matched, but the benign ones should not. Therefore, though the implementation of the detection process seems different, the logic of the workflow is the same as in previous designs.

4.4 Evaluation

Table 4.6: The detection performance for our tool and other existing ones on the 17,770 contracts, where #N refers to the number of detections, P% and R% refer to the precision rate and the recall rate among the number of detections, respectively. Note that P%= (#TP of the tool)/#N, and R%= (#TP of the tool)/ (#TP in the union of all tools).

		Reentrancy	Tx.origin	Unexpected Revert	Self Destruct
	#N	162	23	356	18
Slither	P%	9.8%	43.4%	5.8%	16.6%
	R%	32.6%	33.3%	67.7%	42.8%
	#N	28	N.A.	N.A.	N.A.
Oyente	P%	7.1%	N.A.	N.A.	N.A.
	R%	4.1%	N.A.	N.A.	N.A.
	#N	N.A.	44	51	N.A.
Smartcheck	P%	N.A.	33.3%	47.1%	N.A.
	R%	N.A.	56.6%	**77.4%**	N.A.
	#N	797	N.A.	N.A.	N.A.
Securify	P%	1.1%	N.A.	N.A.	N.A.
	R%	18.3%	N.A.	N.A.	N.A.
	#N	119	98	43	20
Vulpedia	P%	**28.5%**	**88.7%**	**48.8%**	**35.0%**
	R%	**69.3%**	**96.6%**	67.7%	**100%**

Experimental Environment. Throughout the evaluation, all the steps are conducted on a machine running on Ubuntu 18.04, with eight core 2.10GHz Intel Xeon E5-2620V4 processors, 32 GB RAM, and 4 TB HDD. We adopt default configurations to run the scanners used in the evaluation.

Tool Implementation. Vulpedia is implemented based on the SLITHER analyzer. We adopt the AST analysis from SLITHER, and we build PDG analysis based on the CFG (control flow graph) and call graph of SLITHER. The vulnerability signatures are implemented as detectors in nearly 1,000 lines of Python code. The demo of our tool can be found at `https://github.com/ToolmanInside/vulpedia_demo`.

Dataset for Tool Evaluation. To take a different dataset from the contracts we used in the empirical study, we get another address list of contracts from Google BigQuery Open Dataset. After removing contracts that were already used in our empirical study, we get the other 17,770 real-world contracts deployed on Ethereum, on which we fairly compare our resulted tool Vulpedia with the version of the scanners: Slither v0.6.4., Oyente v0.2.7, Smartcheck v2.0 and Securify v1.0 that is open-sourced at Dec 2018. The evaluation dataset is opened along with the empirical study dataset at `https://drive.google.com/file/d/1kizsz0_8B8nP4UNVr0gYjaj25VVZMO8C`.

The evaluations are conducted based on a relaxed assumption that the owners of contracts are not malicious. That is, the owners' operations are all deemed as defense behaviors against vulnerabilities. The evaluations aim to answer these RQs:

1. How is the precision of Vulpedia compared with the existing scanners in vulnerability detection?
2. How is the recall of Vulpedia? Can our signature-based method report more vulnerabilities?
3. How is the efficiency of Vulpedia in tool comparison on the datasets?

4.4.1 RQ1: Evaluating the Precision of Tools

As mentioned in subsection 4.3.3, we have learned ten signatures in total for the four types of vulnerabilities. To evaluate the effectiveness of the resulting vulnerable signatures and detection rules, we apply them to the 17,770 newly collected contracts and compare them with the other state-of-the-art detection tools. Details on the performance of each tool are shown in Table 4.6. Note that we manually verify all TPs.

In Table 4.6, we list 280 detection results of Vulpedia, with an average precision of 50.2%, regardless of vulnerability types. In comparison, Slither has an average accuracy of 18.9%; The average precision of Oyente is 7.1%; The average precision of Smartcheck is 40.2%; and the precision of Securify is surprisingly only 1.1%. In the rest of this section, we analyze the false positives of these tools from the perspective of supporting vulnerability signatures.

FPs of Reentrancy. Among the four supported tools except for Smartcheck, Vulpedia yields the lowest FP rate (71.5%) owing to adopting benign signatures for reentrancy. FP rates of other tools are even higher. For example, the FP rate of Securify is 98.9%, as its detection pattern is too general but has not been considered a possible defense against vulnerabilities in code. Slither adopts Rule 2.1 to detect, but it supports no benign signatures — its recall is acceptable, but the FP rate is high. Oyente adopts Rule 2.2 and has no benign signatures — its recall is low due to the strict rule, and its FP rate is also high.

FPs of Unexpected Revert. As summarized in subsection 4.3.3, Slither reports Unexpected Revert vulnerability when a call-in-loop is detected, ignoring the potential false alarms (i.e., low-level call-in a loop). This coarse-detection rule leads to 335 FPs. Smartcheck handles SIG5 but not SIG6 and leads to 27 FPs. In comparison, Vulpedia combines **SIG5** and **SIG6** for integrating the detection rule, yielding the lowest FP rate of 51.2%.

FPs of `Tx.Origin` Abusing. Slither has a strict rule for detecting this type, only checking the existence of `tx.Origin == msg.sender`. This tool also skips the function if a read operation exists to a particular variable msg.sender, ignoring that some of these variables are irrelevant to tx.origin. For the case that tx.origin is compared with an unrelated address variable, Slither reports it as a vulnerability, causing FPs. Comparatively, Smartcheck and Vulpedia manage to include all the identity check cases, which leads to FPs because — accurate symbolic analysis is not adopted in Smartcheck or Vulpedia to suggest whether tx.Origin can be used to replace msg.sender rightly. Hence, the FP rate due to ignoring **SIG8** is higher than that of Vulpedia.

FPs of Self-destruct Vulnerability. Vulpedia has 13 FPs. After inspecting, we find 10 FPs are due to the unsatisfactory handling of **SIG10**. That is, the identity check hides in self-defined modifiers. Function modifiers are overlooked by Vulpedia, causing FPs. Comparatively, Slither only reports three true positives. The reason is that Slither simply says vulnerability when a *SelfDestruct* call is detected. Due to the inconsideration of the potential access controls, Slither performs with less precision than Vulpedia.

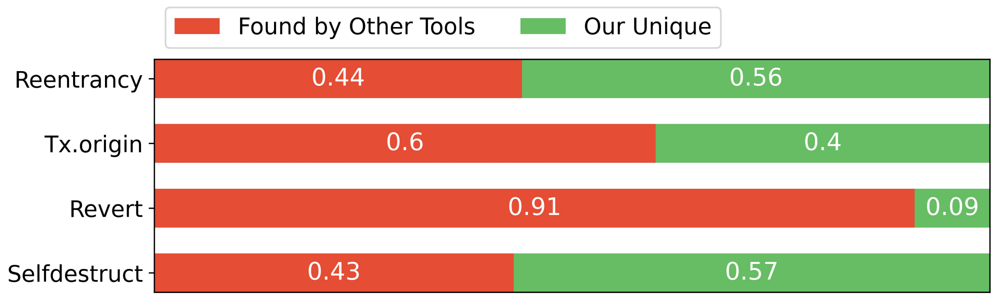


Figure 4.4: Comparing the vulnerabilities only reported by Vulpedia with vulnerabilities reported by other tools. “Our Unique” means those only found by Vulpedia.

Answer to RQ1: Vulpedia performs best in evaluations of precision among tools. In detecting `tx.origin` vulnerability, Vulpedia outperforms the second-best tool by 45.3% (88.7% - 44.3%). The high precision performance is because Vulpedia adopts effective benign signatures to remove false reports.

4.4.2 RQ2: Evaluating the Recall of Tools

In Table 4.6, in most cases, Vulpedia yields the best recall except on unexpected revert, where R% for Smartcheck is 77.4% and R% for Vulpedia is 67.7%. Based on the vulnerable signature abstracted in the empirical study, we expect Vulpedia can find more similar vulnerable candidates. A comparison between vulnerabilities *only* found by Vulpedia (denoted by green bars) and vulnerabilities found by other tools (represented by red bars) is shown in Figure 4.4.

Recall of Reentrancy. In this vulnerability, Vulpedia performs best by reporting 69.3% vulnerabilities. Among all TPs, Vulpedia finds 56% unique TPs that are missed by other evaluated tools. We find that the other three tools commonly fail to consider the user-defined function `transfer()`, not the built-in payment function `transfer()`. For the example in Code 9, Slither and Securify miss it as they mainly check the external call for low-level functions (e.g., `send()`, `value()`) and built-in `transfer()`, ignoring user defined calls. Oyente does not report this example, as it fails in the balance check according to Rule 2.2. Comparatively, Vulpedia detects this vulnerability, as we have a vulnerable signature with a high code similarity with this example. Notably, though Vulpedia has the best recall of 69.3%, it misses 30.7% TPs. This is because reentrancy has many forms, and our vulnerable signature is insufficient to cover those TPs.

Recall of Unexpected Revert. In this vulnerability, the performance of Vulpedia is slightly worse than Smartcheck (77.4%). Specifically, Vulpedia only reports 9% unique TPs, while other tools find 91% TPs. The TPs missed by Vulpedia (reported by Smartcheck) are due to the incompleteness of our vulnerable signature **SIG6**. The signature requires a *ControlDep* after *Call*. However, the *ControlDep* is unnecessary when the *Call* is a high-level call (e.g., user-defined function call) because assertion operations are already integrated with high-level calls. Therefore, the signature causes FNs.

Recall of `Tx.Origin` Abusing. For this type, 96.6% TPs are found by Vulpedia — almost all TPs are found by Vulpedia. Additionally, Vulpedia reports 40% unique TPs, which other tools miss. The reason is that we match the identity check of `Tx.Origin` in self-defined modifiers, which is commonly overlooked by other tools.

Recall of Self-destruct Abusing. For this type, all vulnerabilities (100%) are found by Vulpedia. Comparatively, Slither only reports 42.8% vulnerabilities. 57% of TPs are only found by Vulpedia. The rationale of TPs missed by Slither is that Slither skips the function if the function is only accessible to internal calls (i.e., set visibility to `internal`). These functions are, however, prone to being exploited by internal calls. Therefore, they should not be overlooked. Vulpedia leverages **SIG9** to match vulnerability candidates, so we have better recall performance.

Answer to RQ2: Vulpedia performs best on detection recall. Except for Unexpected Revert, Vulpedia outperforms other tools on three vulnerabilities. This leading performance is because our abstracted vulnerable signatures can represent the essence of most vulnerabilities.

4.4.3 RQ3: Evaluating the Efficiency

On Dataset for Empirical Study. In Table 4.7, Slither takes the least time (only 156 *min*) in detection. Smartcheck and Vulpedia have comparable detec-

Code 9 A real case of reentrancy. This is a TP for Vulpedia but an FN for Slither, Oyente, and Securify.

```
contract Alice {
    ...
    function aliceClaimsPayment(bytes32 _dId, uint _amount,
    ↪ address _addr) external {
        require(deals[_dId].state==DS.Initialized);
        ...
        deals[_dId].state = DS.PaymentSentToAlice;
        if (_addr == 0x0) {msg.sender.transfer(_amount);}
        else {
        ERC20 token = ERC20(_addr);
        assert(token.transfer(msg.sender, _amount)); }
    }
}
```

Table 4.7: The time (min.) of vulnerability detection for each scanner on 76,354 and 17,770 contracts.

Dataset	Slither	Oyente	Smartcheck	Securify	Vulpedia
76,354	156	6,434	641	N.A.	883
17,770	52	1,352	141	8,859	295

tion times (500∼1000 *min*). They are essentially the same type of technique — pattern-based static analysis. In practice, they may differ in performance due to implementation differences, but still, they are significantly faster than Oyente, which applies symbolic execution. Compared with other dynamic analysis or verification tools (i.e., MYTHRILL and SECURIFY that cannot finish in three days for the 76,354 contracts), Oyente is quite efficient. Notably, the signature abstraction time of Vulpedia is not included in the detection time, as it could be done offline separately. Since signature abstraction is analogical to rules formulation, it is not counted in the detection time.

On Dataset for Tool Evaluation. On the smaller dataset, we observe a similar pattern of time execution — Slither is the most efficient, Oyente is the least efficient (except Securify), and Smartcheck and Vulpedia have comparable efficiency. Notably, Securify can finish the detection on 17,770 contracts, but it takes significantly more time than other tools. The performance issue of Securify arises due to the conversion of EVM IRs into datalog representation and then the application of the verification technique. Oyente is also less efficient, relying on symbolic execution for analysis. Vulpedia should be comparable to Smartcheck and Slither, as all three use rule-based matching analysis. The extra overheads of Vulpedia, compared with Slither and Smartcheck, are signature-based code matching.

Answer to RQ3: Vulpedia outperforms Securify and Oyente regarding the detection efficiency on both empirical evaluation and tool comparison. In general, Vulpedia is efficient as a signature-based vulnerability detection tool.

4.4.4 Threats to Validity

In our experiments, we adopt recall rate as a metric, which is a potential threat. Generally, the recall rate indicates the number of TPs divided by the number of all vulnerabilities. However, finding all vulnerabilities (i.e., the ground truth) requires an overwhelming effort. In our study, we evaluate recall performance based on the union of vulnerabilities reported by all tools. Additionally, in the abstraction of signatures, we manually confirm signatures, which may introduce bias. To alleviate this, we repeated our experiments three times. Also, we note that randomness is an inevitable factor in evaluating efficiency.

We repeat the experiments five times and record the average values. Besides, the abstracted signatures are prone to introducing incompleteness. To alleviate this, we implement our methods on top of Slither, facilitating our signature abstraction from PDGs.

4.5 Discussions

4.5.1 The Relaxed Security Assumption

The experiments and comparisons are all conducted based on the relaxed security assumption. That is, we assume the operations of the contract owner are not malicious behaviors. We follow this assumption because the security design is more strict than ordinary contracts when the contract is designed for industry needs. In fact, existing successful contracts (e.g., e-voting, NFT) have been audited by experts to be protected from rogue owners. To avoid our tool being blindly used by users and developers, this assumption should be pointed out.

4.5.2 The Weakness of Vulpedia

In this section, we discuss the improvement of the weakness of Vulpedia found in our experiment practice. In our view, involving manual efforts brings biases, and the biases may affect the effectiveness of the tool. However, Vulpedia relies on manual efforts, mainly in the two steps: 1) manually confirm the reports of existing tools in our empirical study. We add man-powers in this step because the existing static tools have severe limitations and produce a large number of false reports. Due to Ren et al. [111], the Slither tool has a false positive rate of over 70%. If the false reports are not removed from all reports, the dataset cannot be correctly labeled, and our signature abstraction is infeasible. 2) We manually integrate the vulnerable signatures and benign ones into vulnerability detection rules. In this step, we use manual efforts to filter out ineffective signatures. This is due to the lack of a smart contract vulnerability benchmark. If we have a benchmark, we can replace the man-powers in this step and filter out ineffective signatures by running testing on the benchmark.

4.6 Chapter Conclusion

We propose Vulpedia, a static analyzer based on abstracted signatures, in this study. We address one essential challenge: the manually predefined detection rules can be obsolete. To this end, we first conduct an empirical study for signature abstraction. We leverage state-of-the-art scanners to detect vulnerabilities in our training dataset. Based on their results, we propose a method to cluster similar contracts and abstract vulnerable and benign signatures. After we collect all signatures, we conduct comparative evaluations with state-of-the-art tools. The results show that Vulpedia performs the best precision on four types of vulnerabilities and leads recall on three types of vulnerabilities with significant efficiency performance.

CHAPTER

FIVE

STUDY ON ADAPTING DATA-DRIVEN METHODS IN SMART CONTRACT TESTING

5.1 Introduction

Ethereum has been at the forefront of most rankings of blockchain platforms in recent years [112]. It enables the execution of programs, called smart contracts, written in Turing-complete languages such as Solidity. Smart contracts increasingly receive more attention, e.g., with over 1 million daily transactions since 2018 [113].

At the same time, smart contracts-related security attacks are also rising. According to [114–116], 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 [117]. Additionally, as shown by Zou *et al.* [76], over 75% of developers agree that smart contract software has a much higher security requirement than traditional software. Considering the close connection between smart contracts and financial activities, the security of smart contract security predominantly affects the stability of 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., [25, 26, 28, 54, 57, 118]) 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 [31, 32, 66, 119, 120]) avoid false alarms by dynamically executing the traces. Their limitation is that there can often be many program traces to execute. Thus smart strategies must be developed to test the program selectively traces 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 [30]).

While existing efforts have identified an impressive list of vulnerabilities, one crucial category, i.e., cross-contract vulnerabilities, has been largely overlooked. 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 2.1 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 [32] (version 1.0) and sFuzz [31] (version 1.0), both missed this vulnerability because they are limited to analyzing two contracts at a time.

Given a large number of cross-contract transactions in practice [121], 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 consider all sequences and interleaving of function calls from multiple contracts.

As fuzzing techniques practically run programs and barely produce false positive reports [32, 68], adopting fuzzing in cross-contract vulnerability detection is preferred. However, we need other techniques to practically guide fuzzers to detect cross-contract vulnerabilities due to efficiency concerns. Previous works (e.g., [122], [123]) have evidenced the advantages of applying the machine learning method for improving the efficiency of vulnerability fuzzing in C/C++ programs. Unlike static rule-based methods, the ML model-based method requires no prior domain knowledge about known vulnerabilities. It can effectively reduce the large search space for covering more vulnerable functions. In smart contracts, existing works (e.g., ILF [75]) 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 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 [124] (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 [111, 125], the existing analysis tools suffer from high false positive rates (e.g., Slither [25] and Smartcheck [54] 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 on the reported vulnerabilities in contracts, and we further train an 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 [126]. Based on this, the overall bias from using a specific tool to identify potentially vulnerable contract functions can be reduced.

Specifically, xFuzz provides multiple ways of reducing the enormous search space. First, xFuzz is designed to leverage an ML model to identify the most vulnerable functions. That is, an ML model is trained to filter most of the benign functions while preserving most of the vulnerable functions. During the training phase, the ML models are trained based on a training dataset containing program codes 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 [127]. 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. 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 their suspiciousness scores, which are 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 [30] and two state-of-the-art dynamic analyzers, i.e., ContractFuzz [32] and sFuzz, on widely-used open-dataset ([128], [129]) 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. Fifteen 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. xFuzz is publicly available at `https://github.com/ToolmanInside/xfuzz_tool`.

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 reduce the search space for exploitable paths significantly, achieving well-trained ML models with a recall of 95%
- We performed a large-scale evaluation and 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 [130] for public access.

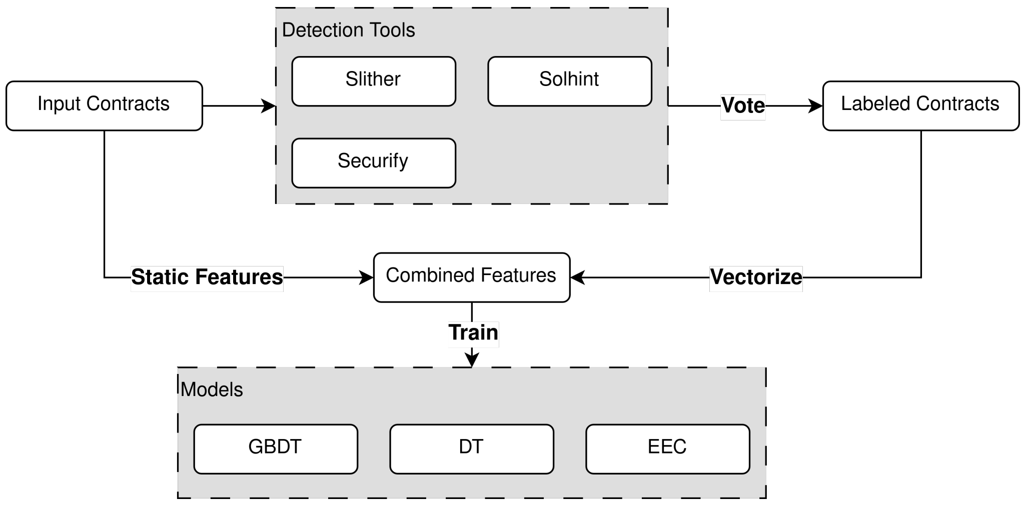


Figure 5.1: The machine learning training phase of xFuzz framework.

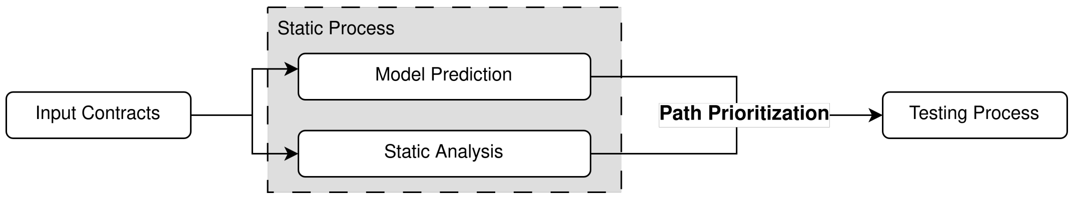


Figure 5.2: The guided fuzzing phase of xFuzz framework.

5.2 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 the 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 addresses the above challenges, as shown in Figure 5.1 and Figure 5.2. Generally, the framework can be divided into two phases: *machine learning model training phase* and *guided fuzzing phase*.

5.2.1 Machine Learning Model Training Phase

In previous works [131, 132], fuzzers are guided by static information (e.g., control flow graphs, call graphs, and data dependency) to traverse particular branches and then find flaws. These methods are limited to prior knowledge of vulnerabilities and 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 the ML model can reduce bias introduced by manually defined detection rules.

We collect training data, engineer features, and evaluate models in this phase. First, we employ the state-of-the-art Slither, Securify, and Solhint to detect vulnerabilities in the dataset. Next, we collect their reports to label contracts. The contract that gains at least two votes is labeled as vulnerability. After that, we engineer features. The input contracts are compiled into bytecode and then vectorized into vectors by Word2Vec [127]. 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 [133], EasyEnsembleClassifier [81] and Decision Tree), among which we select the best one.

5.2.2 Guided Testing Phase

In the guided testing phase, contracts are input to the pre-trained models to obtain predictions. After that, the vulnerable contracts are analyzed and pinpointed. To address challenge **C2**, the functions that are predicted as suspicious ones. Then we use call-graph analysis and control-flow-graph analysis to construct a 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 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 to be 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.

5.3 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 5.3.1 and introduce feature engineering in Section 5.3.2, followed by candidate model evaluation in Section 5.3.3.

5.3.1 Data Collection

SmartBugs [129] and SWCRegistry [134] are two representatives of existing smart contract vulnerability benchmarks. However, their labeled data is scarce, and the available amount is insufficient to train a good model. Therefore, we download and collect contracts from Etherscan (https://etherscan.io/), a prominent Ethereum service platform. To be representative, we collect a large set of 100,139 contracts for further processing.

Table 5.1: Vulnerability detection capability of voting static tools.

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

The collected dataset is then labeled based on the voting results of the three most well-rated static analyzers (i.e., Solhint [118] v2.3.1, Slither [25] v0.6.9 and Securify [28] v1.0). The three tools are chosen because they are 1) state-of-the-art static analyzers and 2) well-maintained and frequently updated. The detection capability varies among these tools (as shown in Table 5.1). We then vote to label the dataset to eliminate each tool's bias. Note that the two vulnerabilities (i.e., delegatecall and tx-origin) are hardly supported by existing tools. Therefore, we only vote for vulnerable functions on vulnerabilities supported by at least two tools. For reentrancy, the voting results are counted so that the function that gains at least two votes is deemed a vulnerability; for tx-origin, the function is deemed a 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. The above vulnerabilities are manually confirmed by two authors of this paper, who have more than three years of development experience for smart contracts to remove false alarms.

5.3.2 Feature Engineering

Then, both vulnerable and benign functions are preprocessed by Slither to extract their runtime bytecode. After that, Word2Vec [127] is leveraged to transform the bytecode into a 20-dimensional vector. However, as reported in [135], vectors alone are still insufficient for training a high-performance model. To address this, we enrich the vectors with seven additional static features extracted from CFGs. In short, the features are 27 dimensions, of which Word2Vec yields 20, and the other seven are summarized in Table 5.2.

Table 5.2: 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

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 a modifier is less like a vulnerable one. Therefore, we make the modifier 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. They are designed in order to better locate the transfer behavior and narrow the 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 determine whether the function can send ethers to others. Since vulnerable functions often have risky transfer operations, this feature can help filter 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. As their names suggest, these two features are specifically designed for the two vulnerabilities. Note that the features can be easily updated to support the detection of new vulnerabilities. If the new vulnerability shares a similar mechanism with the above three vulnerabilities or is closely related, 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 seven static features are combined with word vectors, forming the input to our ML models for further training.

5.3.3 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 [136, 137]. Our dataset endures an imbalance in the rate of 1:126 for reentrancy, 1:2,502 for delegatecall, and 1:298 for tx-origin. Such an imbalanced dataset can hardly be used for training.

To address the challenge, we first eliminate the duplicated data. We found that 73,666-word vectors are the same as others. These samples differ in source code, but after they are compiled, extracted, and transformed into vectors, they share the same values because most are syntactically identical clones [138] at the 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 [139], data sampling strategies can alleviate the imbalance. However, sampling strategies like oversampling [140] can hardly improve the precision and recall of models because the strategy introduces too much-polluted data instead of real vulnerabilities.

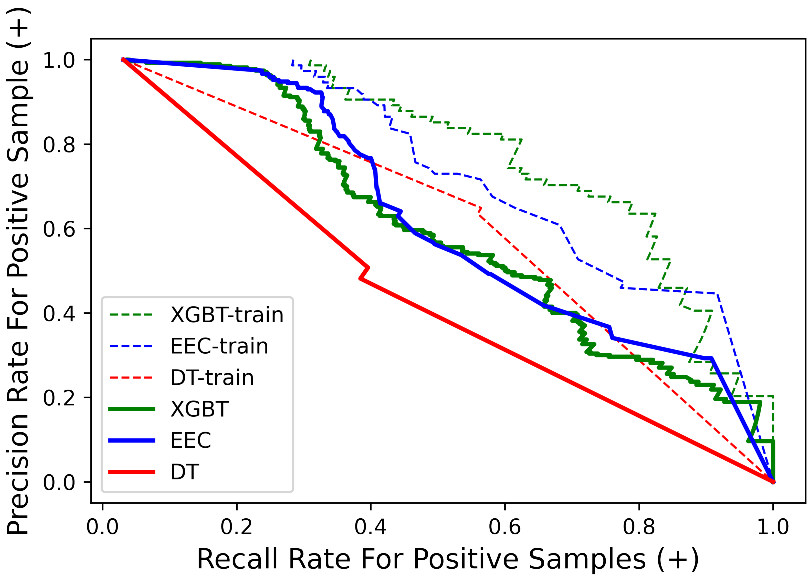


Figure 5.3: The P-R Curve of models. The dashed lines represent performance on the training set, while the solid lines represent performance on the validation set.

We then 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. The models we evaluated include tree-based models XGBT [133], Easy Ensemble Classifier (EEC) [81], Decision Tree (DT), and other representative ML models like Logistic Regression, Bayes Models, and SVMs. The performance of the models can be found in Table 5.3. We find that the tree-based models achieve better precision and recall than others. Other non-tree-based models are biased toward the major class, showing poor classification rates for 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 in Figure 5.3. In this figure, the dashed lines denote models fitting with the validation set, and solid lines indicate fitting with the testing set. Intuitively, model XGBT and EEC achieve better performance with similar P-R curves. However, EEC performs much better than XGBT in the recall. 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

Table 5.3: 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%

that our goal is not to train a very accurate model 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.

5.3.4 Model Robustness Evaluation

To further evaluate the robustness of our selected model and assess how much our model can represent existing analyzers, we conduct an evaluation of comparing the vulnerability detection on unknown datasets between our model and other state-of-the-art static analyzers. The evaluation dataset is downloaded from a prominent third-party blockchain security team[1]. We select smart contracts released in versions 0.4.24 and 0.4.25 (i.e., the majority versions of existing smart contract applications [107]) and remove the contracts 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:

$$CR(t) = (R_m \cap R_t)/R_t \tag{5.1}$$

[1]https://github.com/tintinweb/smart-contract-sanctuary

Table 5.4: The coverage rate (CR) score of ML model on other tools.

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

The results are listed in Table 5.4. 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 the ML model are intersected with static analyzer tools. The coverage rate CR is calculated as listed in Definition 7. The N.A. in the table denotes that the analyzer does not support the detection of this vulnerability.

Our evaluation results show that our tool's reports can cover most other tools' reports. Specifically, the trained ML model can approximate each static tool's capability 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.

5.4 Guided Cross-contract Fuzzing

5.4.1 Guidance Algorithm

The pre-trained models are applied to guide fuzzers in how the predictions are utilized to 1) locate suspicious functions and 2) 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 relatively large, evidenced by the many paths explored by sFuzz (e.g., the 2,596 suspicious functions have 873 possibly vulnerable paths). Thus we propose first to prioritize the path.

Algorithm 4: Machine learning guided fuzzing

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

Code 10 An example of prioritizing paths.

```
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);
    }  }
```

Our guided fuzzing process can be found in Algorithm 4. In this algorithm, we first retrieve the 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. We prioritize the questionable function paths in line 10. The prioritization algorithm can be found in Algorithm 5. Fuzzers will first test the trace with higher priority. Finally, from line 14 to line 21, we select a candidate trace from the prioritized list and employ fuzzers to perform focus fuzzing. The fuzzing process will not end until it reaches a timeout limitation. The found vulnerability will be returned as the final result.

The details of our prioritization algorithm are shown in Algorithm 5. The input to the algorithm is the functions and their corresponding priority scores. The scores are calculated in Algorithm 4. The output of the algorithm is the prioritized vulnerable paths. Specifically, the first step of the algorithm is getting the prioritized function based on the function priority score, as shown in lines 2 and 3. The functions with lower function priority scores will be prioritized. Next, we sort all call paths (whether cross-contract or non-cross-contract call) that correlate to the function, as shown from line 4 to line 6. We pop up the call path with the highest priority and add it to the prioritized path set. The prioritized path set will guide the fuzzer to test the call path in a particular order.

To summarize, the goal of our guidance algorithm is to prioritize cross-contract paths, which are penetrable but usually overlooked by previous practice [31, 32], and further to improve the fuzzing testing efficiency on cross-contract vulnerabilities.

Algorithm 5: 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 $PTR$
```

5.4.2 Priority Score

Generally, the path priority consists of two parts: *function priority* and *caller priority*. The function priority is for evaluating the complexity of the function, and the caller priority is designed to measure the cost of traversing 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 a 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 functions; otherwise, 1 for benign functions. For example, in Code 10, 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 Code 10, the 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 one-dimensionality except for the complex parameters, i.e., they have two dimensionalities. The parameter dimensionality of a function is the sum of parameter dimensionalities. For example, in Code 10, the function `withdraw` and `changeOwner` both have an address and an integer parameter; thus, their dimensionality is 3. The 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:

$$S_{func} = f_s \times (S_C + 1) \times (S_P + 1) \tag{5.2}$$

In this formula, we add 1 to the caller and parameter dimensionality to avoid the overall score being 0. The priority scores in Code 10 are: function `withdraw` = 6, function `changeOwner` = 4, function `logTrans` = 8. The results show that function `changeOwner` has the highest priority because function `withdraw` has two callers to traverse; meanwhile, the function `logTrans` is more difficult to penetrate 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 Code 10, the 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 seeds according to branch distance only, which is not ideal for identifying the three kinds of cross-contract vulnerabilities we focus on in this study. 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. If the function has more than one condition, the distance is the number of statements between the input and the first condition. For example, in Code 10, the condition distance of `changeOwner` is 1, and the condition distance of `logTrans` is 0.

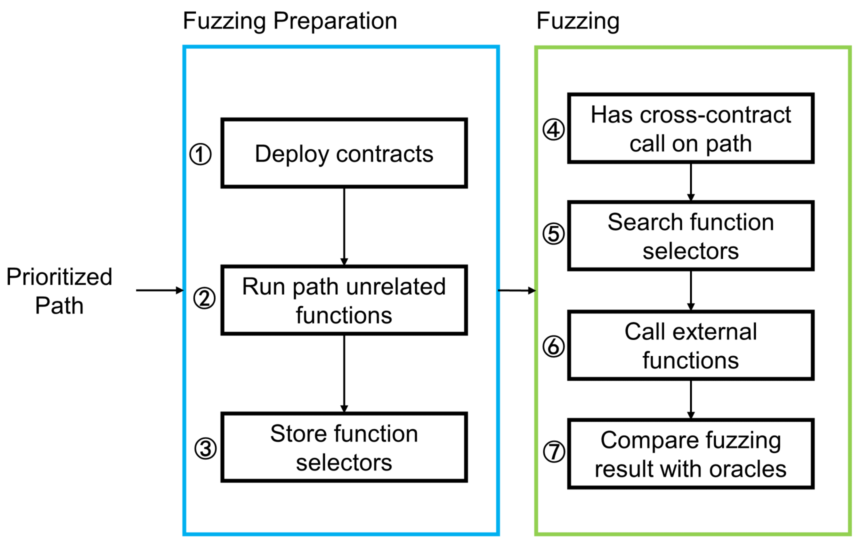
Definition 9 (Caller Priority Score)**.** Given the condition distance *CondDis* and the path condition complexity *Comp*, a path priority score S_{caller} is calculated as:

$$S_{caller} = (CondDis + 1) \times (Comp + 1) \tag{5.3}$$

Finally, the caller priority score is computed based on condition complexity and condition distance, as shown in Definition 9. The complexity and distance add one, so the overall score is not 0. The caller priority scores in Code 10 are: `logTrans` $\rightarrow$ `withdraw` = 1, `changeOwner` $\rightarrow$ `withdraw` = 4. The function `changeOwner` has an identity check at line 6, which increases the difficulty of penetrating. Thus, the other path from `logTrans` to `withdraw` is prior.

5.4.3 Cross-contract Fuzzing

Given the prioritized paths, we utilized cross-contract fuzzing to improve fuzzing efficiency. We implement this fuzzing technique by the following steps: 1) The contracts under test should be deployed on EVM. As shown in Figure 5.4, 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



Fuzzing Preparation
Fuzzing
Prioritized Path
① Deploy contracts
② Run path unrelated functions
③ Store function selectors
④ Has cross-contract call on path
⑤ Search function selectors
⑥ Call external functions
⑦ Compare fuzzing result with oracles

Figure 5.4: The cross-contract fuzzing process.

do not appear in the input prioritized paths. We run them first to initialize the state variables of a contract. 3) We store the function selectors that appeared in all contracts. The function selector is the unique identity recognizer of a function. It is usually encoded in 4-byte hex code [141]. 4) The fuzzer checks whether there is a cross-contract call. If not, the following steps, step 5 and step 6, will be skipped. 5) The fuzzer automatically searches local states to find the correct function selectors and then directly triggers 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.

5.5 Evaluation

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

For the baseline comparison, xFuzz is compared with the state-of-art fuzzer sFuzz [31], a previously published testing engine ContractFuzzer [32] and a static cross-contract analysis tool Clairvoyance [30]. The recently published tool Echidna [66] relies on manually written testing oracles, which may lead to different testing results depending on the developer's expertise. Thus, it is not compared. Other tools (like Harvey [68]) are not publicly available for evaluation and therefore 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 the 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 do the machine learning models and the path prioritization contribute to reducing the search space?

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

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

5.5.1 Dataset Preparation

Our evaluation dataset includes smart contracts from three sources: 1) datasets from previously published works (e.g., [128] and [129]); 2) smart contract vulnerability websites with a good reputation (e.g., [134]); 3) smart contracts downloaded from Etherscan. The dataset is carefully checked to remove duplicate contracts with the dataset used in our machine learning training. Specifically, the **DataSet1** includes contracts from previous works and famous websites. After removing duplicate contracts and toy-contract (i.e., those not deployed on real-world chains), we collect 18 labeled reentrancy vulnerabilities. Our **Dataset2** includes contracts downloaded from Etherscan to enrich the evaluation dataset. 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 two most popular versions of Solidity [107]). Ultimately, 7,391 contracts are collected in **Dataset2**. The source code of the above datasets is publicly available on our website [130] so that the evaluations are reproducible, benefiting further research.

5.5.2 RQ1: Vulnerability Detection Effectiveness

We first conduct evaluations on **Dataset1** by comparing three tools ContractFuzzer, sFuzz, and xFuzz. The Clairvoyance is omitted because it is a static analysis tool. For the sake of page space, we present a part of the results in Table 5.5 with an overall summary and leave the whole list available at here[2].

In this evaluation, ContractFuzzer fails to find a vulnerability among the contracts. sFuzz missed three vulnerabilities and outputted nine incorrect reports. Comparatively, xFuzz missed two vulnerabilities and outputted six incorrect reports. The reason for the overlooked vulnerabilities and incorrect reports lies in the difficult branch conditions (e.g., an `if` statement with three conditions), which blocks the fuzzer from traversing vulnerable branches. Note that xFuzz has model guidance to focus on fuzzing suspicious functions and finding more vulnerabilities than sFuzz.

Table 5.5: Evaluations on Dataset1. The ✔ represents the tool successfully finding a 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

While we compare our tool with existing works on publicly available **Dataset1**, the dataset only provides non-cross-contract labels. It 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

[2]`https://anonymous.4open.science/r/xFuzzforReview-ICSE/Evaluation%20on%20Open-dataset.pdf`

on **Dataset2**. To reduce the effect of randomness, we repeat each set 20 times and report the averaged results.

Table 5.6: 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

5.5.2.1 Cross-contract Vulnerability.

The results are summarized in Table 5.6. Note that the "P%" and "R%" represent precision and recall rates, and "#N" represents the number of vulnerability reports. "C.V." means Clairvoyance and "C.F." means ContractFuzzer. Cross-contract vulnerabilities are currently not supported by ContractFuzzer, sFuzz; thus, they report no vulnerabilities detected.

Precision. Clairvoyance managed to find seven true cross-contract reentrancy vulnerabilities. In comparison, xFuzz found nine cross-contract reentrancy, three cross-contract delegatecall, and two 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, all of which are true positives. The reason for 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 [129, 134, 142], and we find 15 are not flagged.

Recall. The nine vulnerabilities missed by Clairvoyance are all the results of misuse of detection rules, i.e., unsound rules filter out the vulnerable contracts. In total, three cross-contract vulnerabilities are missed by xFuzz. A close investigation shows they are missed due to the complex path conditions blocking the input from penetrating the function. We also carefully check false negatives missed by xFuzz, and find they are not reported by CONRACT-FUZZER and sFUZZ as well. While existing works fail to penetrate the complex path conditions, we believe future works can address this limitation.

5.5.2.2 Non-Cross-contract Vulnerability.

The experiment results show that xFuzz also improves the detection of non-cross-contract vulnerabilities (see Table 5.7). 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 is capable of detecting vulnerabilities by finding 209 (149+35+25) vulnerabilities.

Precision. Clairvoyance reports 75 false positives for reentrancy because of 1) the abuse of detection rules and 2) unexpected jump to unreachable paths due to program errors. The 11 false positives of sFuzz are due to the misconceived ether transfer. xFuzz captures ether transfers to locate dangerous calls. However, the ethers from attacker to victim are also falsely captured. The seven false alarms of xFuzz are due to the mistakes of contract programmers by calling a nonexistent function. These calls are, however, misconceived as vulnerabilities by xFuzz.

Table 5.7: 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

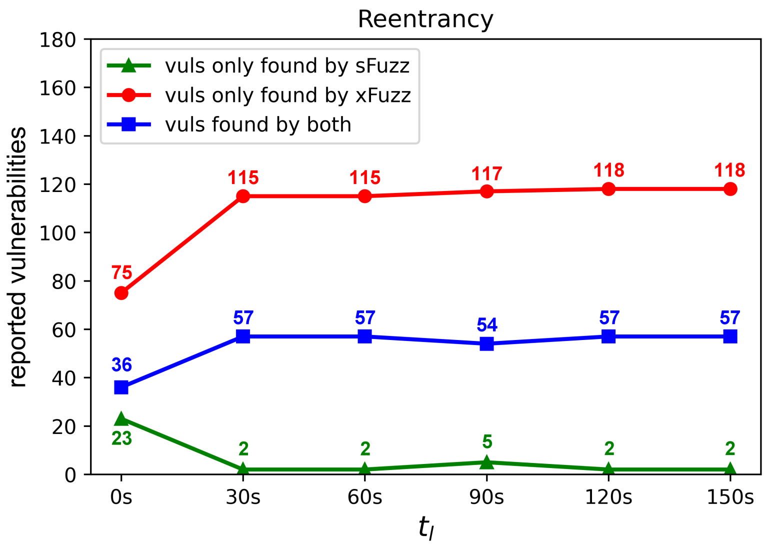


Figure 5.5: Comparison of reported vulnerabilities between xFuzz and sFuzz regarding reentrancy.

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 four evaluated 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.

5.5.3 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 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. We implement xFuzz so that each contract can only be fuzzed for t_l seconds if the ML model

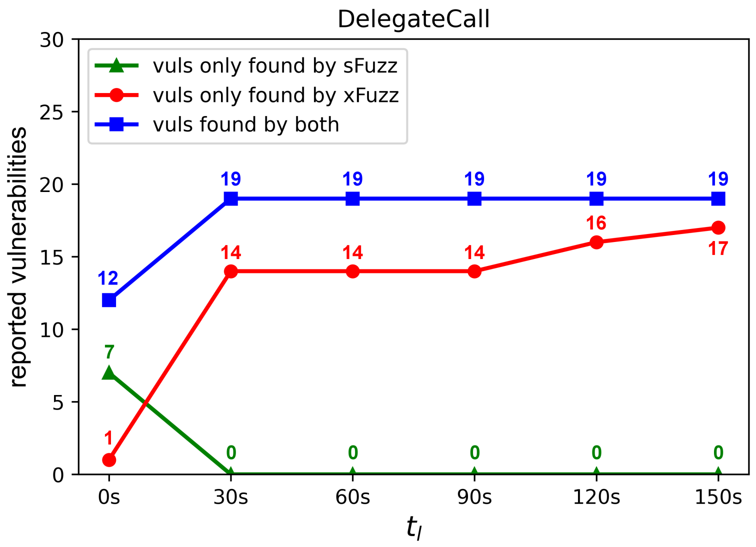


Figure 5.6: Comparison of reported vulnerabilities between xFuzz and sFuzz regarding delegatecall.

considers the contract benign; 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 safely set t_l to be a value smaller than 180 (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 5.5 and Figure 5.6. Note that the tx-origin vulnerability is not included since sFuzz does not support it. 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 tools. We can see that the curves climb/drop sharply at the beginning and then saturate/flatten after the 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 significantly reduces fuzzing time on likely benign contracts (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 expect the vulnerabilities to be mostly found in paths that xFuzz explores first. We, therefore, systematically count the number of vulnerabilities found in the first ten paths that xFuzz explores. The results are summarized in Table 5.8, where column "Top 10" shows the number of vulnerabilities detected in the first ten paths explored.

Table 5.8: 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

The results show that xFuzz finds 152 (out of 172) reentrancy vulnerabilities in the first ten explored paths. In particular, the number of found vulnerabilities in the first ten 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 ten explored paths. The results thus clearly suggest that path prioritization allows us to focus on relevant paths effectively, which has practical consequences 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.

5.5.4 RQ3: Detection Efficiency

Table 5.9: 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

Next, we evaluate the efficiency of our approach. We record the time taken for each step during fuzzing, and the results are summarized in Table 5.9. We replay our experiments five times to eliminate randomness during fuzzing 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 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 since the number of paths (even after the reduction by the ML model and path prioritization) is much larger than in the presence of more than two interacting contracts. Note that Clairvoyance is faster than all tools because this tool is a static detector without performing 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.

5.5.5 RQ4: Real-world Case Studies

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

```
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);
}
```

In this section, we present two typical vulnerabilities reported by xFuzz to show why xFuzz works qualitatively. In general, the ML model and path prioritization help xFuzz find vulnerabilities in three ways, i.e., 1) locate vulnerable functions, 2) identify paths from internal calls and 3) 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 Code 11[3], the modifier `internal` limits the access only to internal member functions. The attacker can, however, steal ethers by path `buyOne` → `buyInternal`. Applying

[3]deployed at 0x0695B9EA62C647E7621C84D12EFC9F2E0CDF5F72

xFuzz identifies the vulnerability 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[4] is shown in Code 12. This example is for auditing transactions in the real world and involves over 2,000 dollars. In this example, the 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 by a cross-contract call `registerAudit` → `audContrat`. Eventually, an attacker would be able to steal the ethers in seconds.

Code 12 A cross-contract vulnerability found by xFuzz. This contract is used in auditing transactions in the real world.

```
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 }
```

Real-world Case 3: During our investigation of the experiment results, we gain the insights that xFuzz can be further improved in handling complex path conditions. Complex path conditions often lead to prolonged fuzzing time or blocking penetration altogether. We identified three cross-contract and 24 non-cross-contract vulnerabilities that were missed for such a reason.

[4]deployed at 0x165CFB9CCF8B185E03205AB4118EA6AFBDBA9203

Two such complex condition examples (from two real-word false negatives of xFuzz) are shown in Code 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 [53] is tasked to solve these path conditions. That is, a hybrid fuzzing approach that integrates symbolic execution in a lightweight manner will likely improve xFuzz further.

Code 13 Complex path conditions involving multiple variables and values.

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

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

Answer to RQ4: With the help of model predictions and path prioritization, xFuzz can rapidly locate vulnerabilities in real-world contracts. The main reason for false negatives is complex path conditions, which could be addressed by integrating hybrid fuzzing into xFuzz.

5.6 Chapter Conclusion

In this paper, we propose xFuzz, a novel machine-learning guided fuzzing framework for smart contracts, focusing on cross-contract vulnerabilities. We address two critical 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 the future, we will extend our framework with the static approach to support more vulnerabilities.

CHAPTER

SIX

CONCLUSION

This chapter concludes the thesis. First, it describes the contributions that emerged from this work. Then, it highlights some possible directions for future work.

6.1 Summary and Contributions of the Thesis

This thesis uses data-driven approaches to adapt existing techniques in a new testing context. The research results can be summarized as follows:

- I surveyed industries to investigate the urgent challenges and pain points for mobile game testing. Based on this, I developed an automated method to collect and label the game GUI dataset. We also created the GUI widget dataset for mobile games. I further integrated state-of-the-art mobile GUI widget detection methods, together forming the first benchmark designed specifically for GUI widget detection research of mobile games.
- I proposed an approach to abstract vulnerability signatures and compose detection rules to report the vulnerability. The learned rules are

more expressive than the rules of the state-of-the-art scanners, reporting vulnerabilities with better completeness and soundness. On the dataset of smart contracts, our approach yields the best precision on four vulnerabilities and leading recall on three ones, compared with the other state-of-the-art scanners. Experiments show that our approach is efficient in vulnerability detection. The detection speed of our approach contracts is far faster than the state-of-the-art.

- I first attempted to formulate and detected three common cross-contract vulnerabilities. I proposed a novel ML-based approach to significantly reduce the search space for exploitable paths significantly, achieving well-trained ML models with a recall of 95%
- I performed a large-scale evaluation and conduct comparative studies with state-of-the-art tools. Leveraging the ML models, our approach outperformed the state-of-the-art tools by at least 42.8% in terms of recall, meanwhile keeping a satisfactory precision of 96.1%.
- Our approach found 18 cross-contract vulnerabilities. All of them are verified by security experts from our industry partner.

6.2 Possible Directions for Future Work

A first avenue for future work is the design and study of adapting testing techniques in new domains using data-driven approaches. Chapter 3 showed that the data-driven approaches can be applied in multiple domains with the dataset. Chapter 4 also showed the necessary efforts to meet the domain gap. Chapter 5 showed that the well-trained data-driven approach can vastly improve testing effectiveness.

A second avenue for further work is strengthening the existing testing approaches by trying more data-driven approaches. Note that with the appearance of the well-known ChatGPT, the data-driven approach has shown the potential to enhance traditional works in many aspects.

REFERENCES

[1] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[2] Jiaming Ye, Mingliang Ma, Yun Lin, Lei Ma, Yinxing Xue, and Jianjun Zhao. Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures. *Journal of Systems and Software*, 192:111410, 2022.

[3] Xiongfei Wu, Jiaming Ye, Ke Chen, Xiaofei Xie, Yujing Hu, Ruochen Huang, Lei Ma, and Jianjun Zhao. Widget detection-based testing for industrial mobile games. In *Proceedings of the 45th IEEE International Conference on Software Engineering*, pages 1427–1437, 2023.

[4] Jiaming Ye, Ke Chen, Xiaofei Xie, Lei Ma, Ruochen Huang, Yingfeng Chen, Yinxing Xue, and Jianjun Zhao. An empirical study of gui widget detection for industrial mobile games. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1427–1437, 2021.

[5] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[6] Ken Shirriff. The programming error that cost Mt Gox 2609 bitcoins. `https://www.righto.com/2014/03/the-programming-error-that-cost-mt-gox.html`, 2014. Online; accessed 29 January 2014.

[7] Ivana Vojinovic. 10 of the Biggest Data Breaches in History. `https://dataprot.net/articles/biggest-data-breaches/textData0individuals.`, 2023. Online; accessed 29 January 2023.

[8] Alfred Ng. How China uses facial recognition to control human behavior. `https://www.cnet.com/news/politics/in-china-facial-recognition-public-shaming-and-control-go-hand-in-hand/`, 2022. Online; accessed 29 January 2022.

[9] OpenAI. chatGPT of OpenAI. `https://openai.com/blog/chatgpt`, 2022. Online; accessed 29 January 2022.

[10] Luciano Baresi and Mauro Pezze. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148(1):89–111, 2006.

[11] Google. American fuzzy lop. `https://lcamtuf.coredump.cx/afl/`, 2018.

[12] LLVM. a library for coverage-guided fuzz testing. `https://llvm.org/docs/LibFuzzer.html`, 2022. Online; accessed 29 January 2022.

[13] Atif M Memon and Myra B Cohen. Automated testing of gui applications: models, tools, and controlling flakiness. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1479–1480, 2013.

[14] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in gui testing of android applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 559–570, 2016.

[15] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18, 2017.

[16] Kevin Moran, Mario Linares Vásquez, and Denys Poshyvanyk. Automated gui testing of android apps: from research to practice. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 505–506, 2017.

[17] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

[18] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. Owl eyes: Spotting ui display issues via visual understanding. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 398–409, 2020.

[19] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: old fashioned or deep learning or a combination? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1202–1214, 2020.

[20] Adrian. Intersection over union. `https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/`, 2018.

[21] NCC Group. Decentralized Application Security Project (or DASP) Top 10 of 2018. `https://dasp.co/`, 2019. Online; accessed 29 January 2019.

[22] ethereum. Solidity document. Website, 2019. `https://solidity.readthedocs.io/en/v0.4.24/contracts.html?highlight=fallback`.

[23] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

[24] ConsenSys Diligence. Ethereum Smart Contract Best Practices:Known Attacks. `https://consensys.github.io/smart-contract-best-practices/known_attacks/`, 2019. Online; accessed 29 January 2019.

[25] Josselin Feist, Gustavo Grieco, and Alex 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)*, pages 8–15. IEEE, 2019.

[26] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC CCS*, pages 254–269, 2016.

[27] ethervm. Ethereum virtual machine opcodes. `https://ethervm.io/`, 2019. Online; accessed September 2019.

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

[29] Tai D Nguyen, Long H Pham, and Jun Sun. sguard: Towards fixing vulnerable smart contracts automatically. *arXiv preprint arXiv:2101.01917*, 2021.

[30] Xue Yinxing, Ma Mingliang, Lin Yun, Sui Yulei, Ye Jiaming, and Peng 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.

[31] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.

[32] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018.

[33] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. From lost to found: Discover missing ui design semantics through recovering missing tags. *Proceedings of the ACM on Human-Computer Interaction*, pages 1–22, 2020.

[34] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. Gallery dc: Design search and knowledge discovery through auto-created gui component gallery. *Proceedings of the ACM on Human-Computer Interaction*, pages 1–22, 2019.

[35] Steven P Reiss, Yun Miao, and Qi Xin. Seeking the user interface. *Automated Software Engineering*, pages 157–193, 2018.

[36] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. Wireframe-based ui design search through image autoencoder. *ACM TOSEM*, pages 1–31, 2020.

[37] Thomas D White, Gordon Fraser, and Guy J Brown. Improving random gui testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 307–317, 2019.

[38] Farnaz Behrang, Steven P Reiss, and Alessandro Orso. Guifetch: supporting app design and development through gui search. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 236–246, 2018.

[39] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83, 2011.

[40] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. Actionnet: Vision-based workflow action recognition from programming screencasts. In *2019 IEEE/ACM 41st ICSE*, pages 350–361. IEEE, 2019.

[41] Dayi Lin, Cor-Paul Bezemer, and Ahmed E Hassan. Studying the urgent updates of popular games on the steam platform. *Empirical Software Engineering*, pages 2095–2126, 2017.

[42] Saiqa Aleem, Luiz Fernando Capretz, and Faheem Ahmed. Critical success factors to improve the game development process from a developer's perspective. *Journal of Computer Science and Technology*, pages 925–950, 2016.

[43] Gabriel Lovreto, Andre T Endo, Paulo Nardi, and Vinicius HS Durelli. Automated tests for mobile games: An experience report. In *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 48–488, 2018.

[44] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 772–784, 2019.

[45] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactions on pattern analysis and machine intelligence*, pages 1904–1916, 2015.

[47] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[48] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *arXiv preprint arXiv:1506.01497*, 2015.

[49] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[50] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37, 2016.

[51] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 845–854, 2017.

[52] Thomas F Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. Learning design semantics for mobile apps. In *Proceedings of the*

31st Annual ACM Symposium on User Interface Software and Technology, pages 569–579, 2018.

[53] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[54] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *WETSEB*, pages 9–16, 2018.

[55] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1678–1694. IEEE, 2020.

[56] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677. IEEE, 2020.

[57] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, 2018.

[58] Llvm language reference manual. `https://blog.sigmaprime.io/solidity-security.html`, 2019. Online; accessed 29 January 2019.

[59] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *CAV 2015*, pages 343–361, 2015.

[60] Octopus. `https://github.com/quoscient/octopus`, 2019. Online; accessed 29 January 2019.

[61] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL*, 2(POPL):48:1–48:28, 2018.

[62] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of*

the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018, pages 653–663, 2018.

[63] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[64] ConsenSys. Mythril. `https://github.com/ConsenSys/mythril-classic`, 2019. Online; accessed 29 January 2019.

[65] ConsenSys. Mythx. `https://mythx.io/`, 2019. Online; accessed 29 January 2019.

[66] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020.

[67] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In *Proceedings of the ISSTA 2019, Beijing, China, July 15-19, 2019.*, pages 363–373, 2019.

[68] Valentin Wüstholz and Maria 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*, pages 1398–1409, 2020.

[69] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *ACSAC*, pages 201–213. ACM, 2016.

[70] NIST. National vulnerability database (nvd). `https://www.nist.gov/programs-projects/national-vulnerability-database-nvd`, 2019. Online; accessed 29 January 2019.

[71] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo: cross-architecture cross-os binary search. In *FSE 2016*, pages 678–689, 2016.

[72] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *PLDI '14*, pages 349–360, 2014.

[73] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *IEEE Symposium on S & P*, pages 595–614. IEEE Computer Society, 2017.

[74] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. Security assurance for smart contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2018.

[75] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin 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*, pages 531–548, 2019.

[76] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 47(10):2084–2106, 2019.

[77] Hao Xiao, Jun Sun, Yang Liu, Shang-Wei Lin, and Chengnian Sun. Tzuyu: Learning stateful typestates. In *2013 28th IEEE/ACM ASE*, pages 432–442. IEEE, 2013.

[78] Yinxing Xue, Junjie Wang, Yang Liu, Hao Xiao, Jun Sun, and Mahinthan Chandramohan. Detection and classification of malicious javascript via attack behavior modelling. In *Proceedings of the 2015 ISSTA*, pages 48–59, 2015.

[79] Guanhua Yan, Junchen Lu, Zhan Shu, and Yunus Kucuk. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*, pages 164–175. IEEE, 2017.

[80] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural network. In *International Joint Conferences on Artificial Intelligence Organization*, pages 3283–3290, 2020.

[81] Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. Exploratory undersampling for class-imbalance learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, pages 539–550, 2008.

[82] Darrenl. How big is the global mobile gaming industry?. `https://www.visualcapitalist.com/how-big-is-the-global-mobile-gaming-industry/`, 2020.

[83] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

[84] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[85] Google. A lightweight, fast, and customizable android testing framework. `https://developer.android.com/training/testing/espresso`, 2018.

[86] Netease Inc. Poco: A cross-engine ui automation framework. `https://github.com/AirtestProject/Poco`, 2018.

[87] Alibaba Inc. Onetomany: a wireless, non-invasive testing tool for automatic android software testing. `https://github.com/alipay/SoloPi`, 2018.

[88] Yuechen Wu, Yingfeng Chen, Xiaofei Xie, Bing Yu, Changjie Fan, and Lei Ma. Regression testing of massively multiplayer online role-playing games. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 692–696. IEEE, 2020.

[89] Android. Ui/application exerciser monkey. `https://developer.android.com/studio/test/monkey`, 2018.

[90] Unity Inc. Unity documentation. `https://docs.unity3d.com/ScriptReference/GameObject.html`, 2018.

[91] Android. Android adb debug mode. `https://developer.android.com/studio/command-line/adb`, 2018.

[92] Darrenl. A graphical image annotation tool. `https://github.com/tzutalin/labelImg`, 2018.

[93] Satoshi Suzuki et al. Topological structural analysis of digitized binary images by border following. *Computer vision, graphics, and image processing*, pages 32–46, 1985.

[94] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[95] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[96] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. In *arXiv preprint arXiv:1904.07850*, 2019.

[97] Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 5551–5560, 2017.

[98] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

[99] Roman Beck, Michel Avital, Matti Rossi, and Jason Bennett Thatcher. Blockchain technology in business and information systems research. *Business & Information Systems Engineering*, 59(6):381–384, 2017.

[100] Nick Szabo. Smart Contracts: Building Blocks for Digital Markets. `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/`

LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html, 1996. Online; accessed 29 January 2019.

[101] Xiangfu Zhao, Zhongyu Chen, Xin Chen, Yanxia Wang, and Changbing Tang. The DAO attack paradoxes in propositional logic. In *ICSAI 2017*, pages 1743–1746, 2017.

[102] Parity Technologies. The Multi-sig Hack: A Postmortem. https://www.parity.io/the-multi-sig-hack-a-postmortem/, July 20, 2017. Online; accessed 29 January 2019.

[103] Adrian Manning. Solidity Security: Comprehensive List of Known Attack Vectors and Common Anti-patterns. https://blog.sigmaprime.io/solidity-security.html, 30 May 2018. Online; accessed 29 January 2019.

[104] Jeffrey D Unman. Principles of database and knowledge-base systems. *Computer Science Press*, 1989.

[105] Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, 2015.

[106] Etherscan. The ethereum blockchain explorer. https://https://cn.etherscan.com/, 2018. Online; accessed September 2018.

[107] Zhenzhou Tian, Jie Tian, Zhongmin Wang, Yanping Chen, Hong Xia, and Lingwei Chen. Landscape estimation of solidity version usage on ethereum via version identification. *International Journal of Intelligent Systems*, 37(1):450–477, 2022.

[108] D. Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 01 1977.

[109] David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, 1978.

[110] Secure smart contract security with transaction-ordering dependence. https://www.nvestlabs.com/2019/03/18/secure-smart-contract-security-with-transaction-ordering-dependence/, 2019. Online; accessed 29 January 2019.

[111] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan 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*, pages 566–579, 2021.

[112] VIKASH KUMAR DAS. Top blockchain platforms of 2020. `https://www.blockchain-council.org/blockchain/topblockchainplatformsof2020that\everyblockchainenthusiastmustknow/`, 2020. Online; accessed September 2020.

[113] Ethereum. Ethereum daily transaction chart. `https://etherscan.io/chart/tx`, 2017. Online; accessed 29 January 2017.

[114] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 2020.

[115] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd ICSE*, pages 530–541, 2020.

[116] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[117] Osman Gazi 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.

[118] Protofire. Solhint. `https://github.com/protofire/solhint`, 2018. Online; accessed September 2018.

[119] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. Ethploit: From fuzzing to efficient exploit generation against smart contracts. In *2020 IEEE 27th SANER*, pages 116–126. IEEE, 2020.

[120] Jianbo Gao, Han Liu, Yue Li, Chao Liu, Zhiqiang Yang, Qingshan Li, Zhi Guan, and Zhong Chen. Towards automated testing of blockchain-based decentralized applications. In *IEEE/ACM 27th ICPC*, pages 294–299, 2019.

[121] Gustavo A Oliva, Ahmed E Hassan, and Zhen Ming Jack Jiang. An exploratory study of smart contracts in the ethereum blockchain platform. *Empirical Software Engineering*, pages 1–41, 2020.

[122] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 60–71. IEEE, 2019.

[123] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017.

[124] Software Testing Help. 7 principles of software testing: Defect clustering and pareto principle. `https://www.softwaretestinghelp.com/7-principles-of-software-testing/`, 2021.

[125] Asem Ghaleb and Karthik 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*, pages 415–427, 2020.

[126] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural network. In *IJCAI*, pages 3283–3290, 2020.

[127] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[128] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan 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*, pages 566–579, 2021.

[129] João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. Smartbugs: A framework to analyze solidity smart contracts. *arXiv preprint arXiv:2007.04771*, 2020.

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

[131] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, 2016.

[132] Will Drewry and Tavis Ormandy. Flayer: Exposing application internals. *WOOT 07, First workshop on offensive technologies*, 2007.

[133] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[134] Smart Contract Security. Smart contract weakness classification registry. `https://github.com/SmartContractSecurity/SWC-registry`, 2019. Online; accessed September 2019.

[135] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 2019.

[136] 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*, pages 103184–103197, 2019.

[137] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, page 85–96, 2016.

[138] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670, 2002.

[139] Joffrey L Leevy, Taghi M Khoshgoftaar, Richard A Bauder, and Naeem Seliya. A survey on addressing high-class imbalance in big data. *Journal of Big Data*, page 42, 2018.

[140] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[141] Smart Contract. Function selector. `https://solidity-by-example.org/function-selector/`, 2021.

[142] Dedaub. Security technology for smart contracts. `https://contract-library.com/`, 2020. Online; accessed 29 January 2020.