5-2018

Automatic vulnerability detection and repair

Siqi MA
Singapore Management University, siqi.ma.2013@phdis.smu.edu.sg

Automatic Vulnerability Detection and Repair

Siqi MA

Singapore Management University
2018

Automatic Vulnerability Detection and Repair

by

Siqi MA

Submitted to School of Information Systems in partial fulfillment of requirements
for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

Robert Huijie DENG (Supervisor / Chair)
Professor of Information Systems
Singapore Management University

David LO (Co-supervisor)
Associate Professor of Information Systems
Singapore Management University

Yingjiu LI
Associate Professor of Information Systems
Singapore Management University

Yongdong WU
Senior Scientist
Institute for Infocomm Research(I2R), A*Star

Singapore Management University
2018

Automatic Vulnerability Detection and Repair

Siqi MA

Abstract

Vulnerability becomes a major threat to the security of many systems, including computer systems (e.g., Windows and Linux) and mobile systems (e.g., Android and iOS). Attackers can steal private information and perform harmful actions by exploiting unpatched vulnerabilities. Vulnerabilities often remain undetected for a long time as they may not affect the typical functionalities of systems. Thus, it is important to detect and repair a vulnerability in time. However, it is often difficult for a developer to detect and repair a vulnerability correctly and timely if he/she is not a security expert. Fortunately, automatic repair approaches significantly assist developers to deal with different types of vulnerabilities. There are lots of work to detect different vulnerabilities, and only few vulnerability repair approaches are proposed to repair certain types of vulnerabilities.

In this dissertation, we first target on one type of vulnerabilities in Android applications, which is cryptographic misuse defects. Cryptography is increasingly being used in mobile applications to provide various security services; from user authentication, data privacy, to secure communications. We propose *CDRep*, which is a novel tool for automatically repairing cryptographic misuse defects. We classify such defects into seven types of misuses, and manually assemble the corresponding fix patterns based on the best practices in cryptographic implementations. CDRep first detects and locates the cryptographic defects. It then automatically repairs the vulnerable application based on the fix patterns that we generated. Such scheme also indicates an inherent limitation that it is tedious to summarize fix pattern manually.

Following the first work, we further explore the feasibility of designing practical scheme to learn fix patterns automatically, which is VuRLE. VuRLE first learns transformative edits and their contexts (i.e., code characterizing edit locations) from

examples of vulnerable codes and their corresponding repaired codes. It then clusters similar transformative edits. Finally, VuRLE extracts edit patterns and context patterns to create several repair templates for each cluster. VuRLE uses the context patterns to detect vulnerabilities, and customizes the corresponding edit patterns to repair them. VuRLE solves the limitations in our first work. It not only generates templates automatically, but also targets on multiple types of vulnerabilities.

Two major contributions are achieved in this dissertation: 1) repair the vulnerabilities by employing the present automatic vulnerability repair schemes; 2) generating repair patterns of vulnerabilities automatically. The proposed repair methodologies are applicable to not only one type of vulnerability, but rather various kinds of vulnerabilities. The proposed schemes are implemented as a prototype, which can be used to automatically repair different vulnerabilities.

However, these proposed approaches focus on repairing the vulnerabilities that exists locally (i.e., at user side). Some other vulnerabilities are caused during the data transmission, such as authentication between client and server. This type of vulnerability can only be detected while analyzing both client code and server code. It is more challenging because some vulnerabilities can only be detected after running the program for several times.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Professor Robert H. DENG, Associate Professor David LO for their guidance in my research, helping me develop strong research skills. Also, they encourage me a lot to be strong and better. I am also very grateful to the other members of my thesis committee, Associate Professor Yingjiu LI and Doctor Yongdong WU, for their guidance and advice in completing my dissertation. Their comments help me clarify my thesis, refine my approach and make me become a more rigorous researcher.

Also, I would like to thank all my teammates, Ke XU, Xiaoxiao TANG, and Siqi ZHAO, for their friendship. In this 5 years research life, their companion makes this period have much more fun. Furthermore, I would like to thank all my co-authors and collaborators: Cong SUN, Teng LI, Junzuo Lai, Baodong QIN and Shaowei WANG, for their indispensable collaboration help.

Finally, I would like to thank my parents, who are always supporting me and encouraging me with all their best wishes. Thanks, mummy & daddy.

List of Publications

Conference Papers

Siqi MA, Ferdian Thung, David LO, Cong SUN, Robert H. DENG. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples. In *Proceedings of the 22nd European Symposium on Research in Computer Security*, Norway, 2017.

Siqi MA, David LO, Teng LI, Robert H. DENG. CDRep: Automatic Repair of Cryptographic Misuses in Android Application. In *Proceedings of the 11th ACM Asia Conference on Computer & Communications Security*, China, 2016.

Siqi MA, Shaowei WANG, David LO, Robert H. DENG, Cong SUN. Active Semi-supervised Approach for Checking App Behavior against Its Description. In *Proceeding of the 39th Annual International Computers, Software & Applications Conference*, Taiwan, 2015.

Ximeng LIU, Hui ZHU, Jianfeng MA, Jun MA, **Siqi MA**. Key-Policy Weighted Attribute based Encryption for fine-grained access control. In *Workshop of IEEE International Conference on Communications*, Australia, 2014.

Journal Papers

Siqi MA, Junzuo LAI, Robert H. DENG, Xuhua DING. Adaptable key-policy attribute-based encryption with time interval. Soft Computing. 21(20), 2017, pp. 6191-6200.

Siqi MA, David LO, Ning XI. Collaborative "many to many" DDoS Detection in cloud. International Journal of Ad Hoc and Ubiquitous Computing. 23(3/4), 2016, pp. 192-202.

Baodong QIN, Robert H. DENG, Shengli LIU, **Siqi MA**. Attribute-Based Encryption With Efficient Verifiable Outsourced Decryption. IEEE Transactions on Information Forensics and Security. 10(7), 2015, pp. 1384-1393.

Jianfeng WANG, Hua MA, Qiang TANG, Jin LI, Hui ZHU, Siqi MA, Xiaofeng CHEN. Efficient Verifiable Fuzzy Keyword Search over Encrypted Data in Cloud Computing. Computer Science and Information System. 10(2), 2013, pp. 667-684.

Jianfeng WANG, Hua MA, Qiang TANG, Jin LI, Hui ZHU, Siqi MA, Xiaofeng CHEN. A New Efficient Verifiable Fuzzy Keyword Search Scheme. Journal of Wireless Mobile Network. 3(4), 2012, pp. 61-71.

Chapter 1

Introduction

1.1 Problem Overview

The widely adaption of applications not only helps user save private information, but also processes data that can only be read by specific people. Keeping private information and data secure is one of the major themes of information security. However, vulnerability becomes a major threat to those applications, including computer applications and mobile applications. By exploiting different vulnerabilities, attackers can steal private information or even perform harmful actions to computer systems. However, vulnerabilities usually survive for a long time. The average lifetime of Android-related vulnerabilities is at least 724 days [47] and the attack on software vulnerabilities usually lasts for 312 days [13]. It is difficult for developers to detect a vulnerability if it is not exploited. Also, vulnerabilities are difficult for developers to repair by themselves, since developers are not security experts and they are unable to repair the vulnerabilities in correct ways. Automatic vulnerability detection and repair approaches significantly help developers and users to deal with vulnerabilities.

The vulnerability detection approaches are usually applied from three perspective: 1) leverage some common features of one type of vulnerability; 2) summarize the pattern from the vulnerable code; 3) extract the code logic from the secure

code that does not contain any vulnerability. The feature-based vulnerability detection methods are usually applied for Intrusion Detection System(IDS), which use the collected common features to build a detection model [7, 67]. However, the false position of most model-based detection methods is high, and it makes the analyst difficult to identify the vulnerability. For the pattern-related tools, researchers analyze a type of vulnerability and design several constrains. Based on the pre-defined constraints, those tools are able to detect the corresponding vulnerability. This approach is usually applied on the network vulnerability [72, 89]. Similar to the pattern-related tools, detection tools that detects based on code logic also need a pre-defined code logic [56]. A significant limitation of those detection approaches is that they require an accurate pre-defined constraints or rules.

The design of vulnerability repair approaches also have some challenges. With plenty of vulnerabilities being detected, only few of them are proposed to be repaired automatically. The repair difficulty comes from the fact that different vulnerabilities have their specific *vulnerability signatures* [45, 15]. Therefore, it requires to learn different vulnerability signatures and creates templates to repair them respectively. The existing vulnerability repair approaches are mainly designed to repair a specific type of vulnerability, such as buffer overflow vulnerability [46], and component hijacking vulnerability [92]. However, different vulnerabilities may occur, which require different patterns. It is impractical to propose an approach for each vulnerability.

To address the above problems, we target on several aspects:

- **Automatic Code Logic Extraction.** How to extract the programming logic?
- **Automatic Rule/Pattern Generation.** How to generate rules for different types of vulnerabilities?
- **Automatic Patch Generation.** How to repair a vulnerability efficiently and accurately?

1.2 Research Objectives

In this dissertation, We propose several approaches to answer the above questions. Based on the programming logic, it is able to extract different repair patterns for different vulnerabilities. These repair patterns are applied to construct effective repair schemes with the assistance of the state-of-the-art technology. We first propose a tool to deal with one type of vulnerability (i.e., cryptographic misuse defects). Next, we target on multiple types of vulnerabilities and design an approach that learns vulnerability repair patterns and repairs those vulnerabilities automatically. The details of these works are introduced as follows.

1.2.1 Cryptographic Misuse Repair

Automatic software repair is a branch of code synthesis. Code synthesis often generates surprising code (i.e., alien code) [54]. To avoid this problem, we first propose to designing a tool to repair cryptographic misuse defects automatically. Cryptographic primitives are widely used to keep users' private information secure, and cryptographic misuses defects are remained unpatched.

To repair a cryptographic misuse defect, we apply this tool on Android applications (called app for short). Our tool can help users repair cryptographic misuse defects to protect their private data. Since our tool is generated from user's perspective, we repair the application at bytecode level. The technical challenge to repair a vulnerability at bytecode level is to handle all the registers assigned to each value. We first summarize correct ways to implement cryptographic algorithms and generate seven repair templates manually. Based on the repair templates, our tool locates those cryptographic APIs, and then identifies misuses that are described in the template.

By evaluating our approach on mobile applications, our results show that the repair scheme is able to be applied on most vulnerable applications. It also reveals some limitations: 1) Manual template generation is tedious and time consuming. To

repair the other vulnerabilities, it requires to generate different templates manually by analyzing plenty of human-written examples. 2) In mobile system, apps can communicate with each other through *Intent*. Our approach assumes each app is isolated.

1.2.2 Multiple Vulnerabilities Repair

In the first work, we generate repair templates to achieve automatic repair. However, manual repair template generation is tedious and time consuming. It is limited to be applied to few types of vulnerabilities. Addressing the limitation of the first work, the second work in this dissertation explores the feasibility of designing practical vulnerability repair scheme with the assistance of existing patched examples. Although many prior efforts [90, 92, 32, 39] have been devoted to repair different vulnerabilities and bugs automatically, but there is still no practical and widely adopted solution up to now. This raises a question on the practicability of adopting an effective pattern generation scheme in vulnerability repair.

In this study, we propose to learn vulnerability repair edits from repair examples. Each example includes a piece of vulnerable code segment and its corresponding repair code segment. Those repair examples are taken as input and edits (i.e., insert, delete, update, move) are learnt to transfer a vulnerable code to its repaired code. For each edit, we also extract its context (i.e., unchanged code that is related to the edits), which is used to locate a vulnerability. We then cluster similar edits into groups. Within those groups, we generate several templates by comparing each pair of edits and replacing variable names. Unlike traditional repair approaches, our approach targets on multiple types of vulnerabilities instead of one, and it is able to generate templates automatically by learning those repair examples. Moreover, our templates are generated semantically, which means that several templates may be generated according to the programming logic for one vulnerability.

We apply our design to existing vulnerabilities, which reveals and identifies sev-

eral limitations. The major limitation is that it requires large amount of examples. Our future analysis indicates that it is possible to overcome these limitations by analyzing program logic.

1.2.3 Authentication Misuse Flaw

To address the limitations mentioned in the first and the second work, we discuss our future research direction in the fifth chapter. The limitations are solved from two perspectives: 1) We analyze secure applications instead of vulnerable applications, since it is difficult to identify whether an application is vulnerable; 2) We extract code logic from the secure code, and generate secure rules to describe the correct way to implement a secure implementation.

As the web applications have been widely used on both mobile platforms and non-mobile platforms, we focus on authentication misuse flaw in our future research. To achieve login scheme, several authentication protocols are applied on Android applications. We first learn the correct ways to implement those authentication protocols and generate the corresponding rules to describe the correct implementation. Any code logic that violates any secure rules is marked as a vulnerable code, that is, this code contains authentication misuses flaw.

1.3 Dissertation Organization

The reminder of this dissertation is organized as follows: Chapter 2 is a literature review which introduces some related researches about vulnerability detection and repair. Chapter 3 describes CDRep as a tool to repair cryptographic misuse automatically. To address the limitations of CDRep, Chapter 4 describes VuRLE that generates repair templates and repair vulnerabilities automatically. Chapter 5 discusses our future research direction on authentication misuses of Android applications. Finally, Chapter 6 summarizes the contributions of this dissertation and discusses a few directions for the future work.

Chapter 2

Literature Review

As vulnerability has become a severe threat to users, lots of vulnerability detection tools are proposed. We summarize the closely related research work from the following aspects. First, we describe vulnerability detection approaches from two perspectives (i.e, non-mobile application and mobile application). Then, we introduce several automatic bug repair tools, which include normal bug repair schemes and vulnerability repair schemes. Finally, we demonstrate the detection schemes to detect zero-day vulnerabilities.

2.1 Vulnerability Detection for non-Mobile Applications

The non-mobile applications include normal software applications, web applications that provide web services, etc. In this section, we only introduce the tools that are applied to the non-mobile application written in C program or Java program.

2.1.1 Buffer Overflow Vulnerability

Most recent detection tools to detect buffer overflow vulnerability [78, 28, 86, 57, 88, 24] perform symbolic execution and data flow analysis. TaintScope [78]

performs dynamic taint analysis to identify potential checksum check points in C program. It achieves the major tasks as follows: 1) Detect checksum test in tested program; 2) Bypass checksum test when fuzzy-testing; 3) Reconstruct input with valid checksum. By running some malformed inputs, it confirms the checksum points. TaintScope is able to found vulnerabilities caused by buffer overflow, integer overflow, null pointer dereference, infinite loop, and double free call. Dowser [28] combines taint tracking, program analysis, and symbolic execution to detect buffer overflow. Instead of analyzing all possible execution paths, Dowser applies spot checks on a small number of code segments that contain buffer overflow vulnerability potentially, and tests them in turn. Code property graph [86] is proposed to detect common vulnerabilities with graph traversals, such as buffer overflow, integer overflow. It combines all basic program analysis schemes (i.e., abstract syntax trees, control flow graphs, and program dependence graphs). Code property graph exposes all information to describe the sources that are controlled by attackers and sensitive operations that are executed. KPSec [57] enables to determine whether a patch brings new vulnerabilities. It performs symbolic execution with static data flow analysis to locate the patch-related code. Based on the patch-related code, KPSec tracks all security points along the path, and then identifies memory-related vulnerabilities (e.g., buffer overflow, memory leaks) by applying multiple security checks.

2.1.2 SQL Injection & Cross-Site Scripting

SQL injection [11, 8, 27, 40, 72] and cross-site scripting (XSS) [74, 41, 58] are usually caused by invalid input sanitization.

Machine Learning related Scheme. Shar et al. [62] propose an approach with hybrid program analysis. Instead of only mining the static code patterns, they also preform dynamic analysis to extract the execution traces of inputs and sanitization functions, respectively. By applying supervised learning and unsupervised learning

method, they are able to build a vulnerability predictor. Shar et al. [61] use static code attributes based on the existing taint analyzer. By analyzing the normal implementation code that is able to avoid SQL injection and XSS vulnerability, they learn code patterns and extract a set of static code attributes. A prediction model can be built based on the static code attributes.

Rule-based Scheme. Sunkari et al. [71] leverage HTTP request analysis to build dynamic rules for the requests of normal web application. They extract the request from every trusted user to summarize a set of attribute-value pairs, which can be used to detect vulnerabilities. Sonewar et al. [69] also collect the web request to generate a fixed query set for static web applications. The static mapping is used to detect vulnerabilities. AMNESIA [27] uses model-based approach to detect illegal queries before they are executed on the database. It uses static analysis to build a model by using legitimate queries, and checks the generated queries against this model. CANDID [11] generates a benign query structure by analyzing benign inputs dynamically, and compares an unknown input with the query structure to detect SQL injection attack. By combining static and dynamic analysis, Lee et al. [40] propose an approach to remove attribute values of SQL queries at runtime, which consists variables in form of string or numeric. It then compares SQL queries with those filtered variables to abnormal queries.

2.2 Vulnerability Detection for Mobile Applications

Some vulnerabilities in mobile applications are different from the vulnerabilities in non-mobile applications, such as component hijacking vulnerability. Component hijacking vulnerability allows an attack to gain unauthorized access to protected or private resources through exported components. The other common vulnerabilities, such as cryptographic misuses and SSL misuses, are being re-implemented in mobile application and services.

2.2.1 Component Hijacking Vulnerability

Detection tools to detect component hijacking vulnerability [50, 55, 42, 80, 10, 43] usually describe a dataflow graph from an entry point to a sensitive sink. CHEX [50] identifies the entry points of an application, and splits the application code into a subset of code according to the identified entry points. CHEX then tracks data-flows crossing and checks the existence enabling hijacking data-flows through dependence graphs. CHEX focuses on finding data-flows between entry points and API calls. To extract more complete flow information, Epicc [55] is proposed to detect inter-component communication (ICC) vulnerability. Since apps are interacted through ICC objects, it specifics a communication across every ICC source to sink, which includes the location of the ICC entry point or exit point. Icc-TA [42] tracks the propagation of context information among components to detect ICC-based privacy leaks. It performs inter-component communication to taint and track the sensitive data.

2.2.2 Cryptographic Misuses

Cryptographic misuses [18, 64, 44, 14, 87] are often caused by inappropriate API usage. CRYPTOLINT [18] performs static analysis to detect common cryptographic flaws in Android application. Six rules are pre-define to achieve a secure cryptographic implementation. CMA [64] extends the implementation rules that are used in CRYPTOLINT, and then builds several models for 13 secure implementation rules. ICryptoTracer [44] is applied on iOS applications. It traces the usage of cryptographic APIs to extract the trace log and analyzes whether the cryptographic API violates the generic cryptographic rules. Braga et al. [14] state the security requirements for an instant message application, and then analyze its implementation of cryptography.

2.2.3 Authentication Protocol Vulnerability

Similar to cryptographic misuses, authentication protocol vulnerability [20, 33, 5, 30, 76] is also caused by incorrect implementation. MalloDroid [20] detects SSL/TLS code that are vulnerable to Man-In-The-Middle Attack (MITMA) in Android apps. It checks the validity of certification, hostname verification, and secure connect usage. Kim et al. [33] analyzes the Pseudo Random Number Generator in Android OpenSSL architecture to identify whether the generated random number is predictable. Alavi et al. [5] perform a study on Android web applications by targeting on several authentication behaviors, such as login, sign up, IP-changing. Different from the previous mentioned authentication protocol vulnerabilities, a code injection vulnerability [30] is proposed, which exists in HTML5-based mobile apps. It causes code injection attack that attacker is able to extract user's information through SMS, Barcode, MP3, etc.

2.3 Automatic Bug Repair

As for automatic bug repair, it is illustrated from two perspectives, normal bug repair and vulnerability (i.e., security bug repair).

2.3.1 Normal Bug Repair

As abundance of defects existing in softwares, repair approaches [39, 82, 38, 32, 53, 85, 37, 36, 6] are proposed to repair software bugs automatically. GenProg [39, 82] and HDRepair [38] perform genetic programming to generate patches. Based on the input program and test cases, GenProg [39] applies mutation and crossover operations to select the most fit individual to repair bugs. HDRepair [38] automatically analyzes bug fix history to infer many graph-based fix pattern, which are used to guide a genetic programming solution to generate high-quality patches. The approach using genetic programming generates non-sensitive patches, relying

on random program mutation operations. This limitation is addressed by PAR [32], which is first proposed to generate patches based on fix patterns. PAR has 10 fix templates generated by fix patterns, which are manually learnt from prior human-written patches. However, manually summarized fix patterns is tedious and time consuming activities. Fixing different vulnerabilities require different fix patterns. It is expensive or even impractical to manually create specific templates or rules for all kinds of vulnerabilities. LASE [53] is further proposed to learn fix patterns automatically. It learns an edit script from two or more repair examples, and then creates a general fix template for a bug. LASE is sensitive to repair examples, that repair examples should be precisely classified to generate the most general pattern.

Recently, a syntax-related program repair tool, ssFix [85] is proposed. ssFix first targets on the suspicious statements that are seems to be incorrect. It then extracts the syntax-related statements. Based on the correct code, ssFix matches the expressions and statements in the correct code with those in the suspicious statements. S3 [37] leverages programming-by-examples methodology to synthesize a better bug repair. It uses the feature with a higher rank from the perspective of syntax and semantic.

2.3.2 Vulnerability Repair

Different from the approaches for normal bug repair, existing vulnerability repair approaches [46, 68, 90, 66, 65, 92, 48, 9, 93] target on one type of vulnerability. Since vulnerability detection and repair are complicated, which require to analyze program flow to identify a vulnerability and a corresponding way to repair it.

Vulnerability Repair for non-Mobile Applications

AutoPaG [46] and FixMeUp [68] perform vulnerability repair approach on vulnerable code. AutoPaG detects an out-of-bound vulnerability and identifies its root cause based data-flow analysis. By considering a manually created official patch, it

generates a patch to repair the out-of-bound vulnerability. FixMeUp indicates conditional statements of a correct access-control check to repair access-control bugs in web applications. It automatically computes an inter-procedural access-control template if a missing access-control check exists. It further transforms the access-control template to repair vulnerable code. Instead of repair the vulnerable code, Yu et al. [90] generate safe inputs to repair vulnerabilities, and DIRA [66] aims to erase malicious packets to repair vulnerabilities. Yu et al. manually constructs input string patterns and attack patterns. Through these patterns, they propose an approach to repair string vulnerabilities in web applications. Based on the input-attack patterns, they are able to compute a safe input, and a malicious input can be converted into a safe input. DIRA focuses on detecting and repair control-hijacking attack by blocking attack packets and repairing component with compromised application's state. It first detects control-hijacking attack, and then identifies malicious network packets that will cause control-hijacking attack. If attack packets are received, it erases the side-effects before the attack happens.

Vulnerability Repair for Mobile Applications

AppSealer [92] defines manually crafted rules for different types of data to repair component hijacking vulnerabilities by using taint analysis. By applying dataflow analysis, it can identify tainted variables, and further repair those variables based on the defined rule. With manually crafted rules, RelFix [48] is proposed to fix resource leak bugs on Android applications. By analyzing call graph, RelFix locates resource leakage. It then generates generates auxiliary variables to trace the resources dynamically to prevent leakage. Different from repairing the vulnerability in source code, Armando et al. [9] prevent a vulnerability, affecting a kernel-level socket (i.e., Zygote socket), in application launching flow. They provide two approaches, i.e., Zygote process fix and Zygote socket fix, to check fork request in Zygote process and reduce linux permissions for the Zygote socket. Instead of creating rules or patterns, Embroidery [93] transplants official patches (CVE source

code patches) of known vulnerabilities, and then rewrites the binary code to implement Android vulnerability patch.

2.4 Zero-day Vulnerability Detection and Repair

Most vulnerability detection and repair techniques are proposed to apply on known vulnerabilities. Some researches are performed on identifying and repair unknown vulnerabilities [77, 60, 81, 91].

MemSherlock [60] identifies unknown memory corruption by detecting malicious payloads. It is applied on source code level. By providing the corruption point in the source code, other consecutive source code, and description of how the malicious input exploit the vulnerability, MemSherlock uses these information to detect unknown vulnerabilities. ShieldGen [81] is able to detect and repair unknown vulnerabilities without any human effort. It has an oracle that can perform zero-day attack. It detects unknown vulnerabilities by input attack data, and then repairs the detected vulnerability based on attack data. Wang et al. [77] propose k-zero day safety, which is a novel security metric to measure how many zero day vulnerabilities are required to compromise a network. It achieves that zero-day vulnerabilities are able to be detected without any measurable information. Zhang et al. [91] proposes a diversity network to improve the resilience of a software system against unknown vulnerabilities.

Chapter 3

CDRep: Automatic Repair of Cryptographic Misuses in Android Applications

3.1 Introduction

This chapter introduces an automatic approach to repair one type of vulnerability (i.e., cryptographic misuse) in Android applications (called apps for short).

Mobile computing has become a fundamental feature in the lives of billions of people, heralding an unprecedented reliance on smart phones and tablets compared to any previous computing technology. With the trend of Bring Your Own Device (BYOD), mobile devices are increasingly used to access and store sensitive corporate information. Thus, many app developers use cryptographic primitives, such as symmetric key encryption and message authentication codes (MACs), to secure communications. However, developers can easily make mistakes in implementing and using cryptography in their mobile applications due to either a lack of cryptographic knowledge or human error, and such mistakes often lead to a false sense of security.

There are a few efforts in the literature investigating the problem of crypto-

graphic misuses in mobile apps. Egele et al. [18] examined if developers use cryptographic APIs in a fashion that provides typical cryptographic notions of security, For example, indistinguishability under chosen plaintext attack (IND-CPA) security and cracking resistance. They found that about 90% of the 12,000 applications in the Google Play marketplace that use cryptographic APIs make at least one mistake. Shuai et al. [64] built a collection of cryptography misuse models, and implemented an automatic misuse detection tool, Crypto Misuse Analyzer (CMA). They found that more than half of the apps they examined suffer from cryptographic misuses. Li et al. [44] designed a tool called iCryptoTracer which traces cryptographic usage in iOS apps, extracts the trace log and judges whether apps have used cryptography correctly. Veracode in 2013 detected cryptographic usage problems in the source codes of mobile apps and concluded that such problems affect 64% of Android apps and 58% of iOS apps [4]. Given the significant portion of mobile apps affected by cryptographic misuses, it is imperative that such misuses be rectified as soon as possible to avert potential attacks, such as brutal force dictionary attack. Unfortunately, it may not be realistic to expect developers who misused cryptography in the first place to do a good job in fixing the misuses because of their lack of cryptographic knowledge or that they are simply unaware of the problem.

Our work aim to repair cryptographic misuses in Android apps automatically. There exist a few efforts in automatically repairing software code in the literature. Most of the previous works have focused on fixing general bugs, such as repairing null pointer dereferences. Goues et al. [39] introduced an approach to repair software programs using genetic programming. Kim et al. [32] proposed a novel patch generation approach by first learning common fix patterns from human-written patches and then generating program patches from these common fix patterns. To our knowledge, very few efforts focus on automatic repair of mobile app vulnerabilities. Recently, Zhang et al. [92] proposed a technique which generates a patch for component hijacking vulnerability in Android apps; they performed static analysis on bytecode to locate vulnerabilities, and then inserted new code to taint

data and track and block dangerous information flow during runtime. Different from Goues et al. and Kim et al., we focus on specific bugs that correspond to cryptographic misuses. Their generic approaches have low success rates (e.g., only 27 out of 119 bugs are successfully fixed by Kim et al.'s approach). In this work, we make use of specialized domain knowledge to fix specialized bugs to achieve a high success rate. Zhang et al.'s approach can also fix specialized bugs with a high success rate, however, they focus on a different kind of vulnerabilities and their approach cannot be used to fix cryptographic misuses that are considered in this chapter.

The automatic repair tool we propose, CDRep (Cryptographic-Misuse Detection and Repair) which automatically detects and repairs misuses of cryptographic APIs at the bytecode level in Android apps. We focus on bytecode level following Zhang et al. [92] since we want to protect users who only have access to the bytecode but not source code of apps. CDRep is designed to repair seven types of misuses identified in [18, 64] and operates in two phases: *detection phase* and *repair phase*. In the detection phase, CDRep locates misuses and classifies them following the light-weight static analysis approach proposed by Egele et al. [18]. In the repair phase, CDRep automatically applies and adapts a set of manually created patch templates to repair a vulnerable program. These patch templates can be created one time and used to repair many vulnerable apps with cryptographic misuses. We apply CDRep on 8,640 real-world Android apps and it detects that 8,582 apps have cryptographic misuses. We manually check a random sample of 1,262 apps from the 8,582 apps and among the 1,262 vulnerable apps, CDRep successfully repairs 1,132 of them.

In a nutshell, the contributions of this work are two-fold:

- We propose and implement CDRep to automatically generate patches to fix cryptographic misuses in Android apps. This is the first effort to repair cryptographic misuses automatically.

- We apply CDRep to 8,640 real-world Android apps. We ask members of

our security research team to evaluate the correctness of the automatic repair. Moreover, we email the repaired apps to their developers to check whether CDRep inadvertently changes behaviors of repaired the apps. Our experimental results show that CDRep is able to repair cryptographic misuses effectively, achieving a successful repair rate of 94.5%. A total of 230 developers responded to our emails and 87.0% of them accepted our patches.

3.1.1 Applications of CDRep

Indeed, the cryptographic misuses could happen due to two reasons:

- Developer lacks the knowledge of cryptography.
- The Android app is developed by an attacker, which means the app is malicious.

In view of the above reason, the cryptographic misuse vulnerability could not be repaired from the developer's perspective. If the developer lacks the knowledge of cryptography, then it might be impossible for developer to repair the cryptography misuses correctly. Further, if the Android developer is an attacker, the developer will definitely leave the vulnerabilities which help the attacker collect users' privacy. These circumstances explain that we are unable to obtain the application source code, namely, the cryptographic misuse could only be repaired on bytecode level.

Handling the repair by users and maintenance companies. CDRep provides a reliable and easy way to repair cryptographic misuses. Users and maintenance companies enable to repair the vulnerability without the source code of an application. Moreover, CDRep provides the standard implementations for different cryptographic approach. They do not need to have any knowledge of cryptography.

Processing the repair for a batch of apps. The minimum overhead helps users and maintenance companies to process a batch of apps. CDRep assures the minimum changes of the app and the changes of the app and the minimum overhead

to process the repair. There exist lots of apps that developers will not upgrade or maintain. However, users might still use them. The maintenance companies could use CDRep to fix the cryptographic misuse of those apps.

3.1.2 Organization

The rest of this paper is organized as follows. Section 3.2 introduces the types of misuses that CDRep intends to detect and repair. Section 3.3 presents the overview of our approach. Section 3.4 elaborates the repair phase of our approach. Experimental results are shown in Section 3.5. Section 3.6 discusses the limitations. Section 3.7 concludes this chapter and describes the future work of our approach.

3.2 Rules of Cryptographic Misuses

In this section, we list the seven security rules that are used in our work, and any application that violates any of those rules cannot be secure [18, 64].

Based on the precisely defined cryptographic algorithms, seven rules are proposed by [18, 64] as follows:

Rule 1: Do not only use electronic codebook (ECB) mode for encryption [84].

Rule 2: Do not use a constant Initialized Vector (IV) for ciphertext block chaining (CBC) encryption.

Rule 3: Do not use constant secret keys.

Rule 4: Do not use constant salts for password-based encryption (PBE).

Rule 5: Do not use fewer than 1,000 iterations for PBE.

Rule 6: Do not use static seeds to seed *SecureRandom*.

Rule 7: Do not use reversible one-way hash (i.e. reversible MD5 message-digest algorithm).

Encryption schemes are used to protect user privacy, ensuring that attackers are unable to extract even a single bit of plaintext from a ciphertext within a reasonable time bound. *Indistinguishability under a chosen plaintext attack* (IND-CPA) is proposed to formalized this notion, and an encryption scheme could be defined as secure if and only if it is IND-CPA secure [18]. However, some encryption mode or wrong implementations make the encryption scheme become non IND-CPA secure, such as using ECB mode and using constant value. Therefore, seven rules are proposed to keep the encryption scheme secure. Based on the seven rules, we defined seven types of misuse, and each misuse violates one of the rules of security. To identify the misuse in a bytecode, we first examine the instruction that is related to the misuse, called *Indicator Instruction*. Then, we locate the instruction that causes the misuse, called *Root Cause Instruction*. We shows seven types of misuses below and their the corresponding example bytecode. The root cause of each example is set in bold.

Misuse 1: Only use ECB mode to encrypt. ECB mode is not IND-CPA secure in symmetric encryption scheme. The bytecode below shows such misuse. According to the indicator instruction, `Ljava/crypto/Cipher;→getInstance`, we can identify that register v1 holds the value of encryption type. Therefore, we are able to find that value of v1 is "AES/ECB/PKCS5Padding[1]" based on the root cause, which means that the developer uses ECB mode for encryption. Due to that ECB is the default encryption mode set by Android, the developer also uses ECB mode if they only define "AES" in their code.

1. **const-string v1, "AES/ECB/PKCSPadding"**
2. `invoke-static {v1}, Ljava/crypto/Cipher;→`
   `getInstance(Ljava/lang/String;)`
   `Ljava/crypto/Cipher`

[1] AES is the cryptography algorithm chosen by developers. PKCS5Padding is the padding scheme

Misuse 2: Using a constant IV in CBC encryption. In CBC encryption scheme, a constant IV will generate a deterministic, stateless cipher, which is not IND-CPA secure. The bytecode snippet below shows such misuse. Instruction, `Ljava/crypto/spec/IvParameterSpec`, is the indicator instruction of this misuse, and we can conclude that register v7 is the IV parameter. However, v7 receives the value from register v10. Thus, v10 holds the original value of the IV, which is set as constant based on the root cause, "1234567898765432". However, the IV should always be set as random based on the CBC encryption construction.

1. `new-instance v7,`
   `Ljavax/crypto/spec/IvParameterSpec;`
2. **const-string v10, "1234567898765432"**
3. `invoke-virtual v10, Ljava/lang/String/;→`
   `getBytes()[B`
4. `move-result-object v10`
5. `invoke-direct {v7, v10},Ljava/crypto/spec/`
   `IvParameterSpec;→<init>([B)V`

Misuse 3: Using a constant secret key. In a symmetric encryption scheme, if an user uses a secret key with insufficient key length, then the attacker can extract the secret key by using brute force dictionary attack. Moreover, if secret key is constant, it will be extracted by using brute force attack. The sample bytecode with a constant secret key is illustrated as below. According to the indicator instruction, `Ljava/crypto/spec/SecretKeySpec`, we are able to define that register v2 holds the value of secret key, and the value of register v2 is "0x0", which is constant.

1. **const/4 v2, 0x0**
2. `invoke-virtual v2, Ljava/lang/String/;→`
   `getBytes()[B`
3. `move-result-object v2`
4. `const-string/jumbo v4, "AES"`
5. `invoke-direct {v3, v2, v4},`
   `Ljava/crypto/spec/SecretKeySpec;→`
   `<init>([BLjava/lang/String;)V`

Misuse 4: Using a constant salt in PBE. According to [18], a randomized salt can make PBE perform better. A constant salt makes the algorithm with salt reduce to an algorithm without salt. According to the indicator instruction, `Ljava/crypto/spec/PBEParameterSpec` in the sample bytecode shown as below, we observe that it uses PBE encryption scheme, and register v2 holds a constant salt value, "0x0".

1. **const/4 v2, 0x0**
2. `new-instance v3, Ljava/crypto/spec/`
   `PBEParameterSpec;→<init>([BI)V`
3. **const/16 v4, 0x64**
4. `invoke-direct {v3, v2, v4},`
   `Ljava/crypto/spec/PBEParameterSpec;`
   `→<init>([BI)V`

Misuse 5: Setting iterations $<$ 1,000 in PBE. Based on the suggestion given by PKCS#5[2], the iteration should be at least 1,000 (i.e. 0x3e8 in hexadecimal). According to the indicator instruction, `Ljava/crypto/spec/PBEParameterSpec` in the sample bytecode

[2]PKCS#5: Password-Based Cryptography Standard, http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-5-password-based-cryptography-standard.htm.

of misuse 4, it describes the situation where iteration is set inappropriately. Register v4 holds the hexadecimal value of iteration, "0x64" (i.e. 100 in decimal).

misuse 6: Using a constant seed to seed *SecureRandom* In [2], they show that seeding *SecureRandom* may be insecure since seeding may cause the instance to return a predictable sequence of numbers. If the same seed is reused, the returned number will become repeatable. The indicator instruction in the code shown as below is `Ljava/security/SecureRandom;→getInstance`. Based on the indicator instruction, we can identify the root cause instruction and conclude that *SecureRandom* is seeded by a constant seed, that is, the value of register p0.

1. `p0, [B`
2. `...`
3. `const-string/jumbo v1, "SHA1PRNG"`
4. `invoke-static {v2, v1},`
   `Ljava/security/SecureRandom;→getInstance`
   `(Ljava/lang/String;Ljava/lang/String;)`
   `Ljava/security/SecureRandom;`
5. `move-result-object v1`
6. **invoke-virtual {v1, p0}, Ljava/security/**
   SecureRandom;→setSeed([B)V

Misuse 7: Using reversible MD5 hash function. Wang et al. [79] have found many collisions in MD5 and created a powerful attack that can efficiently find MD5 collisions. Based on the indicator instruction, `Ljava/security/MessageDigest` in the sample bytecode below, we infer the root cause instruction and identify that register v2 contains the string

of encryption scheme, that is, “MD5”, allowing us to conclude that it uses MD5 hash function.

1. **const-string v2, “MD5”**
2. `invoke-static {v2}, Ljava/security/MessageDigest;→getInstance(Ljava/lang/String;)Ljava/security/MessageDigest`
3. `move-result-object v1`

3.3 Overview of CDRep

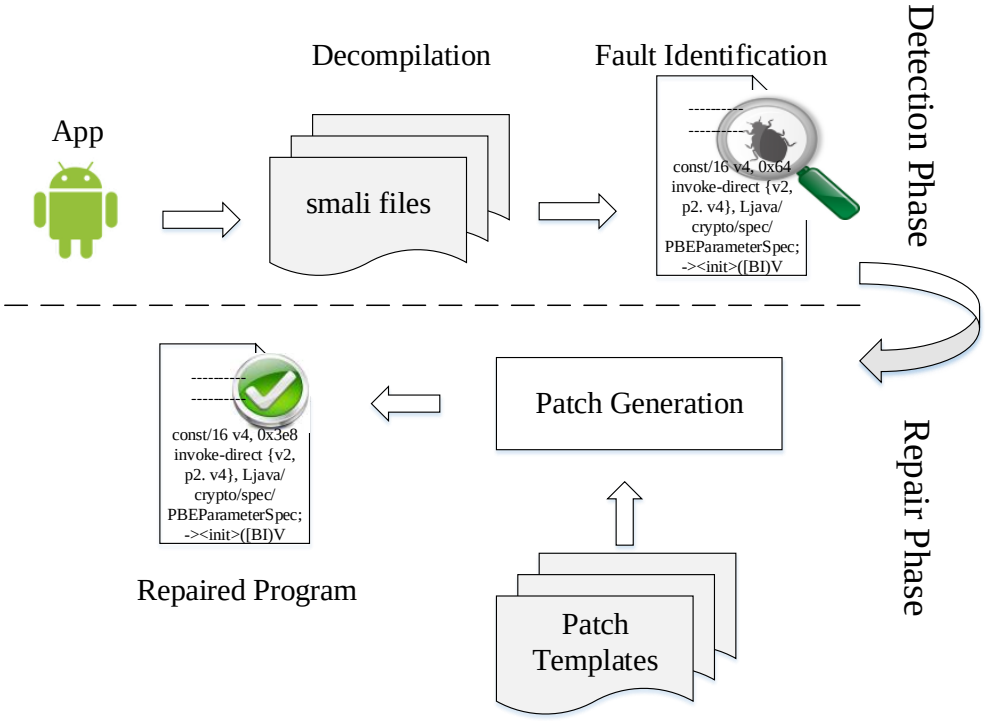


Figure 3.1: Overview of CDRep

In this section, we introduce the overview of our automatic repair technique, CDRep (Cryptographic-Misuse Detection and Repair). Figure 4.1 shows the workflow of CDRep. It has two phases, *detection* and *repair*:

Detection: In this phase, CDRep follows the detection steps of CRYPTOLINT [18], which include *decompilation* and *fault identification*. After decompiling an Android app, in the fault identification phase, CDRep checks if vulnerabilities exist

in the app; if they exist, CDRep identifies vulnerable Java classes as well as their vulnerability types.

Vulnerabilities are found by first locating indicator instructions (see Section 3.2) in the decompiled code. Next, for each indicator instruction, CDRep identifies other instructions that the indicator instruction is data dependent on. CDRep then checks all such instructions to identify root causes that correspond to cryptographic misuses. For each misuse, CDRep records its type and the Java class that contains it. Since this step closely follows CRYPTOLINT, we only briefly describe its intuition. Details are available in the original CRYPTOLINT paper [18]. CRYPTOLINT detects six kinds of vulnerabilities (misuses 1-6); in this work, we add one more vulnerability (misuse 7). The procedure to identify the seventh vulnerability is the same as the one used to identify the other six.

Repair: In this phase, CDRep fixes the vulnerable program by performing a series of program transformations specified in a set of manually created patch templates. Details of this phase is presented in Section 3.4.

3.4 Cryptographic Misuses: Automatic Repair

In this section, we elaborate the repair phase of CDRep. This phase requires a set of manually created patch templates, which we describe in Section 3.4.1. Given a vulnerable Java class and a vulnerability type, CDRep applies a corresponding patch template to repair the class (described in Section 3.4.2).

3.4.1 Patch Templates

We manually create seven patch templates, each for a misuse type. To generate these templates, we take a set of programs with cryptographic misuses and manually fix them. Next, for each pair of correct and faulty program pairs (i.e., with and without cryptographic misuses), we examine code that needs to be added and

Misuse 2:	Using a constant IV for CBC encryption	
Input:	Vulnerable Java class *Target*	
Transformation:	i	Insert ***SecureRandom*** class to the default package of the app
	ii	Insert a new public IV addition ***.field public static ivParams:Ljavax/crypto/spec/IvParameterSpec;***.
	iii	**[Encryption:]** 1 § new-instance Pl_2, Ljavax/crypto/spec/IvParameterSpec; 2 − const Pl_1, * 3 § invoke-virtual {Pl_1}, Ljava/lang/String;->getBytes()[B 4 § move-result-object Pl_1 5 § invoke-direct {Pl_2, Pl_1}, Ljava/crypto/spec/IvParameterSpec;-><init>([B)V 6 + invoke-static {}, SecureRandom; ->gen_ivParams()Ljavax/crypto/spec/IvParameterSpec; 7 + move-result-object Pl_3 8 + sput-object Pl_3, *Target*;->ivParams:Ljavax/crypto/spec/IvParameterSpec; 9 + invoke-virtual {Pl_3}, Ljava/lang/Object;->toString()Ljava/lang/String; 10 + move-result-object Pl_3 11 + invoke-virtual { Pl_3}, Ljava/lang/String;->length()I 12 + move-result Pl_3 13 + move Pl_4, Pl_3 14 + .local Pl_4, "iv_length":I 15 + move Pl_5, Pl_4 16 + const/16 Pl_6, 0x10 17 + add-int/lit8 Pl_5, Pl_5, -0x10 18 + invoke-virtual { Pl_3, Pl_5}, Ljava/lang/String;->substring(I)Ljava/lang/String; 19 + move-result-object Pl_3 20 + move-object Pl_1, Pl_3
	iv	**[Decryption:]** 1 § new-instance Pl_2, Ljavax/crypto/spec/IvParameterSpec; 2 − const Pl_1, * 3 § invoke-virtual {Pl_1}, Ljava/lang/String;->getBytes()[B 4 § move-result-object Pl_1 5 § invoke-direct {Pl_2, Pl_1}, Ljava/crypto/spec/IvParameterSpec;-><init>([B]V 6 + sget-object Pl_3 *Target*;->ivParams:Ljavax/crypto/spec/IvParameterSpec; 7 + invoke-virtual { Pl_3}, Ljava/lang/Object;->toString()Ljava/lang/String; 8 + move-result-object Pl_3 9 + invoke-virtual { Pl_3}, Ljava/lang/String;->length()I 10 + move-result Pl_3 11 + move Pl_4, Pl_3 12 + .local Pl_4, "iv_length":I 13 + move Pl_5, Pl_4 14 + const/16 Pl_6, 0x10 15 + add-int/lit8 Pl_5, Pl_5, -0x10 16 +invoke-virtual { Pl_3, Pl_5}, Ljava/lang/String;->substring(I)Ljava/lang/String; 17 + move-result-object Pl_3 18 + move-object Pl_1, Pl_3

Figure 3.2: Patch template for misuse 2: this template fix the misuse that use a constant IV for CBC encryption

removed to transform the faulty program to the correct one. We then generalize the added and removed code as a generic patch. A generic patch consists of a series of code *transformations*. Each transformation corresponds to a series of code removal and addition given a particular context. To make the patch generic, we replace actual register/variable names, with *placeholders*. We also replace each constant value

with a wildcard character ("*") that can match any constant.

Figure 3.3 presents a sample template for transforming a Java class *Target* containing cryptographic misuse 2, i.e., it uses a constant IV for CBC encryption. The template contains 4 transformations: i, ii, iii, iv. Transformation i specifies the insertion of the bytecode of java.security.SecureRandom class to the app (if it does not exist). Transformation ii specifies the insertion of a field **IvParameterSpec** to *Target*. Transformations iii and iv specifies code additions (marked by "+") and code deletions (marked by "-") along with a context (marked by "="). Each transformation specifies that whenever a piece of code matches with the context, the lines of code marked by "-" will be replaced with the lines of code marked by "+". In the two transformations, we have placeholders (e.g., Pl_1, ... Pl_6 in transformation iii) and wildcard characters (e.g., "*" at line 1 of transformation iii).

It is worth mentioning that cryptographic algorithms always appear in pairs (i.e. encrypt and decrypt). Transformation iii is to fix the encryption method in the vulnerable class and transformation iv fixes the decryption method. In transformation iii, we match for code `Ljava/crypto/spec/IvParameterSpec` to locate indicator instruction (line 5). Then, we replace code that is the root cause of the misuse (line 1) with code that generates the randomized value (line 6-20). The randomized value is stored in the field **ivParams** (line 8). Then we check the length of the randomized value (line 11). Due to that the length is longer than the required length, we only take the sub-length of the randomized value (line 15-18). Finally, the sub-length of randomized value is transformed to the placeholder of IV. Similar to transformation iii, we also match the indiator instruction `Ljava/crypto/spec/IvParameterSpec` first in transformation iv (line 5). Then, we locate the root cause in line 1 and replace it with codes that are used to generate the randomized value. Instead of generating a randomized value, we extract the value from the field **ivParams** in line 6. We take the same steps as tranformation iii to check the length of randomized value and take the required length of randomized value (line 9-16).

Table 3.1: Patch overview

Misuse	Patch Overview
1	Using CTR mode.
2	using a randomized IV for CBC encryption.
3	Using a randomized secret key.
4	Using a randomized salt in PBE.
5	Setting iterations = 1,000.
6	Calling *SecureRandom.nextBytes()*.
7	Using SHA-256 hash funtion.

Another sample template for transforming a Java class *target* containing cryptographic misuse 5, i.e., it sets iterations $<$ 1,000. The template only has one transformation, **modification**. However, the modification transformation should be applied on both encryption and decryption method. We first match the code `Ljava/crypto/spec/PBEParameterSpec;→<init>` in line 4. Then, we locate the root cause in line 3 and replace it with line 5 to modify the iterations to 1,000.

Misuse 5:	Set iteration < 1,000
Input:	Vulnerable Java class *Target*
Transformation:	**[Modification]** 1 § new-instance Pl_1, Ljavax/crypto/spec/PBEParameterSpec; 2 § sget-object Pl_2, *Target*;->salt:[B 3 − const Pl_3, * 4 § invoke-direct { Pl_1, Pl_2, Pl_3}, Ljava/crypto/spec/PBEParameterSpec; -><init>([BI]V 5 + const/16 Pl_3, 0x3e8

Figure 3.3: Patch template for misuse 5: this template fix the misuse that sets the iterations $<$ 1,000

Due to space constraint, we cannot show all the 7 templates. A brief description of these templates is given in Table 3.1. A complete description is available in our technical report [3]. Although the generation of these patch templates is manual, it is a one-time cost and the patch templates can be reused to automatically fix many cryptographic misuses.

3.4.2 Patch Generation

In this step, CDRep takes a vulnerable Java class along with a misuse type as inputs, and generates a patched class. To generate the patched class, CDRep picks the corresponding patch template, runs a series of program transformations specified in the template, and replaces placeholders with actual register/variable names.

Given a transformation, CDRep matches the lines of code marked by "=" and "-" in the vulnerable Java class. In the process, the *mappings* between placeholders and actual variable/register names are identified. The lines of code marked with "-" are then replaced with the lines of code marked with "+". Placeholders in these newly added lines of code are then replaced with actual register names based on the mappings that are identified earlier. Other placeholders in the newly added code that do not appear in the mapping are replaced with new variable/register names that do not appear in the vulnerable Java class.

An automatic code fix example for misuse 2 (i.e., using a constant IV for CBC encryption) is shown in Figure 3.4. According to the transformation iii given in the template, Figure 3.3, CDRep first matches the line of code marked by "=" (i.e., lines 1, 3, 4, 5 in the vulnerable code of Figure (a)). Then, CDRep performs mapping between the placeholders and the actual variable/register shown in Figure (b). It is apparently that register v10 is mapped to placeholder Pl_1, which is the root cause. Register v7 is mapped to placeholder Pl_2, which represents the IvParamterSpec. After mapping the actual registers, CDRep replaces the placeholders that are mapped with the actual registers. For those placeholders that are not mapped with any actual registers, we replace them with other available registers (i.e., v1, v2 shown in Figure 3(c)).

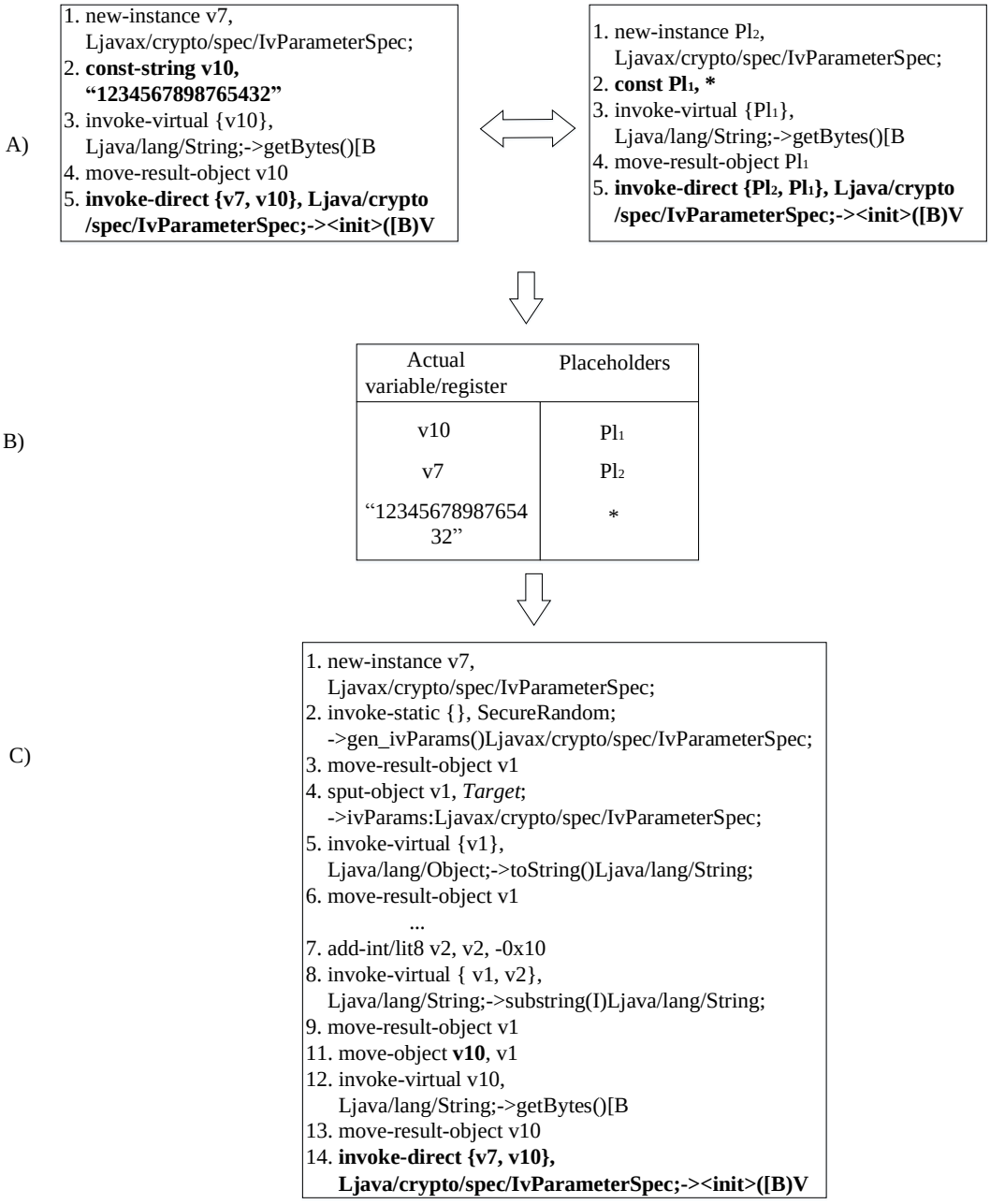


Figure 3.4: Fix procedure for misuse 2: it uses a constant IV for CBC encryption. A) shows the vulnerable code with misuse 2, and the template of misuse 2. B) describes the mapping procedure between the actual variable/register extracted from the vulnerable code and placeholders given in the template. C) is the fixed code by replacing the placeholders by the actual registers that are mapped

3.5 Experiment

In this section, we present the details and results of our experiments that evaluate the performance of CDRep. Our experiments are designed to answer the following questions:

Table 3.2: CDRep: Detection result

Misuse Type	# of Apps	Percentage	Google Play	SlideMe
Rule 1	887	10%	402	485
Rule 2	979	11%	379	600
Rule 3	882	10%	357	525
Rule 4	7	0.08%	4	3
Rule 5	10	0.1%	7	3
Rule 6	235	2%	17	218
Rule 7	5582	65%	1359	4223

RQ1 (Success rate) How many misuses can CDRep repair successfully?

RQ2 (Runtime) What is the average time needed for CDRep to generate a patch?

RQ3 (Size) What is the average increase in size of repaired apps?

RQ4 (Failed cases) Why can't some apps be repaired successfully?

3.5.1 Experiment Setup

Dataset. To evaluate CDRep, we crawled apps from two app stores, Google play[3] and SlideMe[4] (a third-party store). In total, we collected 8,640 free apps (2,114 apps from Google Play, and 6,526 apps from SlideMe), sampled from all categories. Since some sensitive categories (e.g., finance, retail, etc.) are more likely to use cryptography algorithms, we sample more applications from these sensitive categories than others (in a ratio of 5.5 to 1).

Detected Cryptographic Misuses. CDRep performs both detection and fix of cryptographic misuses. Table 3.2 shows the number of cryptographic misuses detected by CDRep across the seven misuse types.

Experiment Design. We evaluate the effectiveness of our approach from three aspects: acceptance rate, patching speed, and size of repaired apps. Acceptance rate evaluates whether our patches are acceptable by security experts and app developers.

[3]Google Play Store: https://play.google.com/store?hl=en
[4]Third-Party store: SlideMe (http://slideme.org/)

Patching speed evaluates the efficiency of our approach; if our approach takes a long time to complete, users are less likely to use it. Size of repaired apps is also an important factor that affects usability; if the size of the patched app increases too much, users are less likely to use it.

To measure acceptance rate, we ask our security research team and application developers to examine the repaired programs. Our research team can examine whether the repaired implementations of the cryptographic functionalities are correct. However, they will not be able to conclude whether our patch inadvertently modifies any other behaviours of the app in a bad way. Thus, we also email the repaired apps to their corresponding developers to get feedback on the app behaviours. To measure patching speed, we simply measure the average time that our approach takes to generate a patch. To measure repaired app size, we measure the percentage of increase in app size after an app has been patched.

To measure acceptance rate, manual inspection (performed by our security research team and app developers) is needed. Since this inspection is a time consuming process, and many apps suffer from cryptographic misuses (see Table 3.2), it is not possible to check all of the apps that we have repaired (especially for apps that exhibit misuse 1-3, and 7). Thus, except for misuse 4-6 (for which we evaluate all repaired apps), for each other misuse type, we randomly sample apps for manual inspection. For misuse 1, 2, 3, and 7, we select 100, 110, 100, and 700 apps respectively. We vary the number of apps selected for each misuse type, based on the number of apps with cryptographic misuses of that type (we pick around 12% of apps of a particular misuse type).

3.5.2 RQ1: Success Rate

In this section, we measure how many vulnerable apps are repaired successfully. To make it easier for cryptographers and developers to examine the patch, we not only give them the original vulnerable app and the repaired app that we have

Table 3.3: Success Rate

	# of Selected apps	Team Acceptance	# of Developer Response	Developer Acceptance
Misuse 1	100	91(91%)	21	13(61.9%)
Misuse 2	110	92(83.6%)	16	10(62.5%)
Misuse 3	100	83(83%)	23	18(78.2%)
Misuse 4	7	5(71.4%)	3	2(66.7%)
Misuse 5	10	10(100%)	4	4(100%)
Misuse 6	235	212(90.2%)	20	15(75%)
Misuse 7	700	700(100%)	143	138(96.5%)
Total	1262	1193(94.5%)	230	200(87.0%)

repacked, but also provide the bytecode of the vulnerable and repaired apps. In addition, we describe the misuses in the app, and explain why the cryptographic code in the app is not secure.

Table 3.3 presents the acceptance result of our repaired apps. Overall, our research team accept more than 94.5% of the repaired apps. Considering the email responses, 87% of the repaired program are accepted, which means that the app behaviors are not impacted by the repaired program. According to the result, the patch for misuse 5 and misuse 7 are better than the other types. Our repaired apps are not accepted by all the developers, we explain the reasons in the following section (Section 3.5.4).

3.5.3 RQ2 and RQ3: Runtime and Size

The average runtime needed by our approach to identify a misuse and generate a patch, excluding decompilation time, is only about 19.3 seconds. The bulk of the cost is in the generation of a patch which on average takes 14.6 seconds.

The increase in the size of the patched apps is negligible. Table 3.4 shows the average increase in the size of patched apps for different misuse types. Across the 7 apps the average increase in size is only 0.667% of the original app size.

Table 3.4: Average Patch Overhead of different misuse type

Misuse Type	Overhead
Misuse 1	0.749%
Misuse 2	0.640%
Misuse 3	0.632%
Misuse 4	0.742%
Misuse 5	0.634%
Misuse 6	0.526%
Misuse 7	0.748%

3.5.4 RQ4: Unsuccessful Cases

From Table 3.3, there are apps that are not repaired successfully by our approach. We discuss the main causes as follows:

Popular libraries. For some apps, developers may call popular libraries. CDRep identifies some misuses that exist in these libraries. For example, several MD5 misuses occur in the classes that are provided by Google, that are, several classes in the "com.google.android.gms.*" package. Although we have repaired those misuses, some app developers rejected our changes since they still prefer to use the standard classes provided by Google.

Incomplete repair: CDRep assumes that each method only contains code that uses one cryptographic scheme. For cases where this assumption does not hold (i.e., a method contains code that uses multiple cryptographic schemes), CDRep could only repair misuses of the first cryptographic scheme. We find that a few apps define more than one encryption scheme in a single method, which causes the patch generated by our approach to be incomplete.

Incomplete decompilation: We use *apktool* to decompile vulnerable apps. However, we find that some apps with complex behaviours cannot be decompiled well. Moreover, some apps reject decompilation. For such cases, CDRep cannot generate patches.

3.6 Limitations

In this section, we discuss the threats to validity. Aside from the limitations corresponding to the unsuccessful cases highlighted in Section 3.5.4, there are a few other limitations of our approach and its evaluation:

Focus on Android. CDRep is only able to detect and fix cryptographic misuses involving cryptographic classes that come with the Android API. An app may use other third-party cryptographic libraries or implement their own. CDRep is not able to detect and fix cryptographic misuses for such apps. To detect these misuses, there is a need to create new templates. This effort will pay off if the third party cryptographic libraries are used by many Android apps.

Focus on Free Apps. In our experiment, we only evaluate the effectiveness of C-DRep on free apps. These apps might not be representative of paid apps. The implementations of paid applications could be different from those of free apps and these differences may impact the effectiveness of our approach. In the future, we plan to expand our study to evaluate the effectiveness of CDRep on paid apps.

Focus on the Interaction. For some apps, they upload user's data to their server instead of keeping it locally. CDRep only ensures that an app could work normally if it processes encryption and decryption on the client side. It might break if this app shares the cryptographic parameters with its server, once we modify the cryptographic method the client side.

3.7 Conclusion and Future Work

In this chapter we propose a approach, CDRep, to automatically repair vulnerable apps with cryptographic misuses. Given a vulnerable Android app, we first perform static analysis to locate the misuse and identify the misuse type. Then,

based on the misuse type, we apply a suitable patch template and adapt it to the vulnerable program by replacing register placeholders in the template with actual register names. Finally, we perform an optimization step to remove dead code. To evaluate CDRep, we crawled 8,640 real-world Android apps and use CDRep to identify cryptographic misuses and repair them. Out of the repaired apps, we randomly pick 1,262 of them for manual inspection (by security experts and app developers). The evaluation results show that CDRep can automatically repair the vulnerable apps effectively – it is able to repair 94.5% of the 1,262 vulnerable apps with an average patch generation time of merely 19.3 seconds.

There are several aspects for future work. CDRep aims to repair the cryptographic misuse by using static analysis at bytecode level. However, detection with static analysis is not complete.

Detect Self-Written Encryption/Decryption class. In the detection phase of CDRep, we detect the cryptographic misuse by using the pre-defined cryptographic APIs that Java provided (e.g., Cipher.getInstance). However, some developers might prefer to call the cryptographic function written by themselves instead of calling the existing cryptographic APIs. CDRep is unable to detect the self-written encryption/decryption class.

Identify Constant Variable. We adopt backward data analysis to identify the constant variable. However, it could only match the constant variable when it is defined in the function. In some circumstances, value of the variable is not set in the function, and it is assigned by the heap during runtime.

We will extend CDRep by applying hybrid analysis (i.e., static analysis and dynamic analysis). Static analysis enables to extract the cryptographic usage from the code level, and dynamic analysis could capture the code behaviors at runtime. It helps detect the self-written encryption/decryption class and provide a more completed data flow graph.

Chapter 4

VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples

4.1 Introduction

This chapter repairs vulnerabilities in computer system instead of Android system. In computer system, vulnerability is also a severe threat, which is difficult for a developer to detect and repair a vulnerability. It motivates researchers to explore practical design to detect and repair different kinds of vulnerabilities in computer system, such as cross-site scripting (XSS) [78], component hijacking vulnerability [92], etc. Similar to the previous work, those studies on automatic vulnerability repair typically focus on one type of vulnerabilities. These studies require custom *manually-generated* templates or custom heuristics tailored for a particular vulnerability.

Manually generating repair templates and defining repair rules are tedious and time consuming activities. As technology and computer systems advance, different vulnerabilities may occur and fixing each of them likely requires different repair patterns. Unfortunately, it is very expensive or even impractical to manually create

specific templates or rules for all kinds of vulnerabilities. The above facts highlight the importance of developing techniques that can generate repair templates automatically.

To help developers repair common bugs, Meng et al. [53] proposed LASE that can automatically generate a repair template. LASE automatically learns an edit script from two or more repair examples. However, its inference process has two major limitations. First, it can only generate a general template for a type of bug. However, a bug can be repaired in different ways based on the *context* (i.e., preceding code where a bug appears in). Second, it cannot learn multiple repair templates from a repair example that involves repair multiple bugs.

Under these limitations, this work explores of designing a practical scheme that is able to generate multiple templates and learns patterns automatically. We design and implement a novel tool, called VuRLE (Vulnerability Repair by Learning from Examples), that can help developers automatically detect and repair multiple types of vulnerabilities. VuRLE can be applied to repair both Android applications and other applications written in Java. VuRLE works as follows:

1. VuRLE analyzes a training set of repair examples and identifies *edit blocks* – each being series of related edits and its context from each example. Each example contains a vulnerable code and its repaired code.

2. VuRLE clusters similar edit blocks into groups.

3. Next, VuRLE generates several repair templates for each group from pairs of highly similar edits.

4. VuRLE then uses the repair templates to identify vulnerable code.

5. VuRLE eventually selects a suitable repair template and applies the transformative edits in the template to repair a vulnerable code.

VuRLE addresses the first limitation of LASE by generating many repair templates instead of only one. These templates are put into groups and are used collec-

tively to accurately identify vulnerabilities. VuRLE also employs a heuristics that identifies the most appropriate template for a detected vulnerability. It addresses the second limitation by breaking a repair example into several code segments. It then extracts an edit block from each of the code segment. These edit blocks may cover different bugs and can be used to generate different repair templates. This will result in many edit blocks though, and many of which may not be useful in the identification and fixing of vulnerabilities. To deal with this issue, VuRLE employs a heuristics to identify suitable edit blocks that can be generalized into repair templates.

We evaluate VuRLE on 279 vulnerabilities from 48 real-world applications using 10-fold cross validation setting. In this experiment, VuRLE successfully detects 183 (65.59%) out of 279 vulnerabilities, and repairs 101 of them. This is a major improvement when compared to LASE, as it can only detects 58 (20.79%) out of the 279 vulnerabilities, and repairs 21 of them.

The rest of this paper is organized as follows. Section 4.2 presents an overview of our approach. Section 4.3 elaborates the learning phase of our approach and Section 4.4 presents the repair phase of our approach. Experimental results are presented in Section 4.5. Section 4.6 concludes the paper and discusses our future work.

4.2 Overview of VuRLE

In this section, we introduce how VuRLE repairs vulnerabilities. Figure 4.1 shows the workflow of VuRLE. VuRLE contains two phases, **Learning Phase** and **Repair Phase**. We provide an overview of working details of each phase below.

Learning Phase. VuRLE generates templates by analyzing edits from repair examples in three steps (Step 1-3).

1. **Edit Block Extraction.** VuRLE first extracts *edit blocks* by performing Ab-

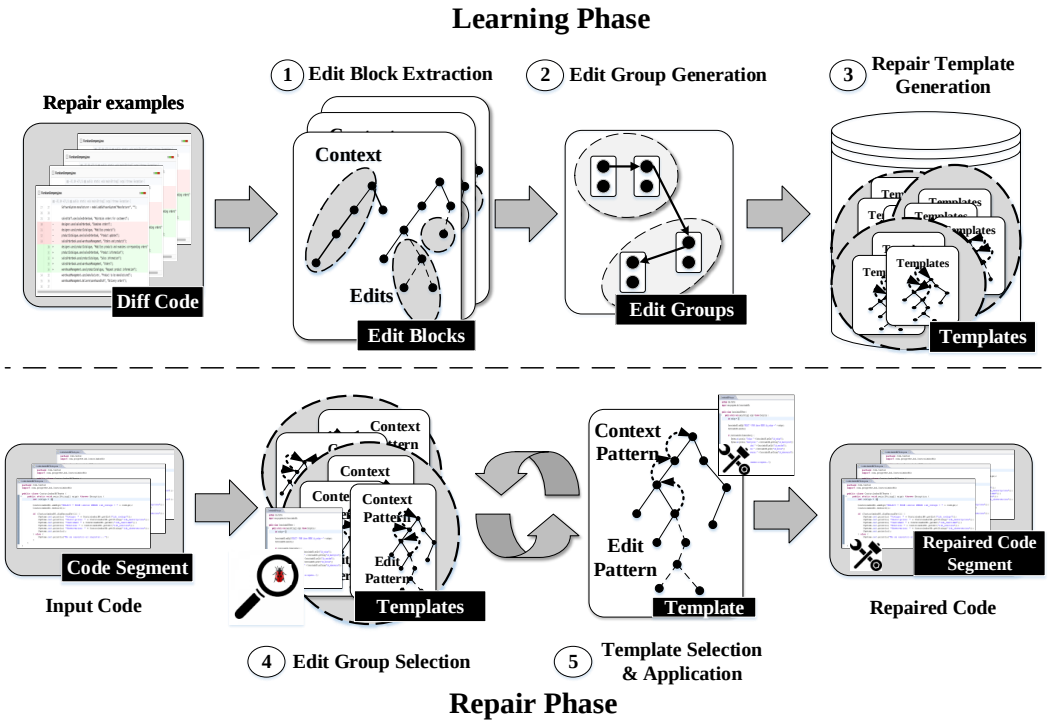


Figure 4.1: Workflow of VuRLE: 1) VuRLE generates an edit block by extracting a sequence of edit operations and its context. 2) VuRLE pairs the edit blocks and clusters them into edit groups 3) VuRLE generates repair templates, and each contains an edit pattern and a context pattern. 4) VuRLE selects the best matching edit group to detect for vulnerabilities 5) VuRLE selects and applies the most appropriate repair template within the selected group.

stract Syntax Tree (AST) *diff* [23] of each vulnerable code and its repaired code in a training set of known repair examples.

The difference between a pair of vulnerable and repaired code may be in several code segments (i.e., contiguous lines of code). For each pair of vulnerable and repaired code segments, VuRLE outputs an edit block which consists of two parts: (1) a sequence of *edit operations*, and (2) its *context*. The first specifies a sequence of AST node insertion, deletion, update, and move operations to transform the vulnerable code segment to the repaired code segment. The latter specifies a common AST subtree corresponding to code appearing before the two code segments.

2. **Edit Group Generation.** VuRLE compares each edit block with the other edit blocks, and produces groups of similar edit blocks.

VuRLE creates these *edit groups* in several steps. First, it creates a graph where each edit block is a node, and edges are added between two edit blocks iff they share the longest common substring [26] of edit operations with a substantial size. Next, it extracts connected components [29] from these graphs. Finally, it applies a DBSCAN [19]-inspired clustering algorithm, to divide edit blocks in each connected component into edit groups.

3. **Repair Template Generation.** In each edit group, VuRLE generates a *repair template* for each pair of edit blocks that are adjacent to each other in the connected component (generated as part of Step 2).

   Each repair template has an *edit pattern* and a *context pattern*. An edit pattern specifies a sequence of transformative edits, while a context pattern specifies the location of the code where the transformative edits should be applied. To create the edit pattern, VuRLE identifies the longest common substring of edit operations in the two edit blocks. To create the context pattern, VuRLE compares the code appearing in the context part of the two edit blocks. To generalize the patterns, VuRLE abstracts concrete identifier names and types appearing in the patterns into *placeholders*.

   The context pattern is used to identify vulnerable code, while the edit pattern is used to repair identified vulnerabilities in the repair phase.

Repair Phase. VuRLE detects and repairs vulnerabilities by selecting the most appropriate template in two steps (Step 4-5). These two steps are repeated a number of times until no more vulnerable code segments are detected.

4. **Edit Groups Selection.** Given an input code and a set of repair templates, VuRLE compares code segments of the input code with edit groups and identifies an edit group that best matches it.

5. **Template Selection & Application.** The most matched edit group may have multiple templates that match an input code segment. VuRLE enumerates the

matched templates one by one, and applies the transformative edits specified in the edit pattern of the template. If the application of the transformative edits results in redundant code, VuRLE proceeds to try the next template. Otherwise, it will flag the code segment as a vulnerability and generates a repaired code segment by applying the transformative edits.

4.3 Learning Phase: Learning from Repair Examples

In this phase, VuRLE processes a set of vulnerability repair examples to produce groups of similar repair templates. The three steps involved in this phase (Edit Block Extraction, Edit Block Group Extraction, and Repair Template Generation) are presented in more details below.

4.3.1 Edit Block Extraction

For each repair example, VuRLE uses Falleri et al.'s GumTree [21] to compare the AST of a vulnerable code and its repaired code. Each node in an AST corresponding to a source code file can be represented by a 2-tuple: (*Type*, *Value*). The first part of the tuple indicates the type of the node, e.g., VariableDeclarationStatement, SimpleType, SimpleName, etc. The second indicates the concrete value stored in the node, e.g., String, readLine, "OziExplorer", etc.

Using GumTree, VuRLE produces for each repair example a set of edit blocks, each corresponds to a specific code segment in the AST diff between a vulnerable code and its repaired code. Each edit block consists of a sequence of edit operations, and its context. The sequence can include one of the following edit operations:

- **Insert**(**Node** u, **Node** p, **int** k): Insert node u as the k^{th} child of parent node p.

- **Delete**(**Node** u, **Node** p, **int** k): Delete node u, which is the k^{th} child of parent node p.

- **Update**(**Node** u, **Value** v): Update the old value of node u to the new value v.

- **Move** (**Node** u, **Node** p, **int** k): Move node u and make it the k^{th} child of parent p. Note that all children of u are moved as well, therefore this moves a whole subtree.

For each sequence of edit operations, VuRLE also identifies its context. To identify this context, VuRLE uses GumTree to extract an AST subtree that appears in both vulnerable and repaired ASTs and is relevant to nodes affected by the edit operations. This subtree is the largest common subtree where each of its leaf nodes is a node with SimpleName type that specifies a variable that is used in the sequence of edit operations. We make use of the `getParents` method of GumTree to find this subtree.

To illustrate the above, consider Figure 4.2. It shows the ASTs of a vulnerable code segment and its corresponding repaired code segment. Performing AST diff on these two ASTs produces a sequence of edit operations which results in the deletion of nodes V_{12} to V_{17}, and the insertion of nodes R_{12} to R_{21} into the subtree rooted at V_3. It also produces a context which corresponds to the common AST subtree highlighted in gray.

4.3.2 Edit Group Generation

VuRLE generates edit groups in two steps: (1) edit graph construction; (2) edit block clustering. We describe these two steps in detail below.

Edit Graph Construction. VuRLE creates a graph, whose nodes are edit blocks extracted in the previous step. The edges in this graph connect similar edit blocks. Two edit blocks are deemed similar iff their edit operations are similar. To check

for this similarity, VuRLE extracts the longest common substring (LCS) [26] from their edit operation sequences. The two edit blocks are then considered similar if the length of this LCS is larger than a certain threshold T_{Sim}. Each edge is also weighted by the reciprocal of the corresponding LCS length. This weight represents the distance between the two edit blocks. We denote the distance between two edit blocks e_1 and e_2 as $dist(e_1, e_2)$.

Edit Block Clustering. Given an edit graph, VuRLE first extracts connected components [29] from it. For every connected component, VuRLE clusters edit blocks appearing in it.

To cluster edit blocks in a connected component (CC), VuRLE follows a DBscan-inspired clustering algorithm. It takes in two parameters: ε (maximum cluster radius) and ρ (minimum cluster size). Based on these two parameters, VuRLE returns the following edit groups (EGS):

$$EGS(CC) = \{N_\varepsilon(e_i) \mid e_i \in CC \wedge |N_\varepsilon(e_i)| \geq \rho\} \tag{4.1}$$

In the above equation, $N_\varepsilon(e_i)$ represents a set of edit blocks in CC whose distance to e_i is at most ε. Formally, it is defined as:

$$N_\varepsilon(e_i) = \{e_j \in CC \mid dist(e_i, e_j) \leq \varepsilon\} \tag{4.2}$$

The value of ρ is set to be 2 to avoid generating groups consisting of only one edit block. The value of ε is decided by following Kreutzer et al.'s code clustering method [34]. Their heuristic has been shown to work well in their experiments. The detailed steps are as follows:

1. Given an edit graph, VuRLE first computes the distance between each connected edit block. Two edit blocks that are not connected in the edit graph has an infinite distance between them.

2. VuRLE then orders the distances in ascending order. Let $\langle d_1, d_2, ..., d_n \rangle$ be

the ordered sequence of those distances.

3. VuRLE finally sets the value of ε by finding the largest gap between two consecutive distances $d_{\langle j+1 \rangle}$ and $d_{\langle j \rangle}$ in the ordered sequence. Formally, ε is set as $\varepsilon = d_{\langle j^* \rangle}$, where $j^* = argmax_{1 \leq j \leq n}(\frac{d\langle j+1 \rangle}{d\langle j \rangle})$.

To illustrate the above process, Figure 4.3 presents two connected components (CCs), $\{E_1, E_2, E_3, E_5, E_6\}$ and $\{E_0, E_7\}$. VuRLE first orders the distances into [0.12, 0.14, 0.17, 0.25]. It then computes the largest gap between two consecutive distances, and identifies a suitable value of ε, which is 0.17. Based on ε = 0.17 and ρ = 2, VuRLE creates two groups of edit blocks for the first CC: $\{E_1, E_2, E_3\}$, and $\{E_5, E_6\}$. It generates none for the second CC.

4.3.3 Templates Generation

For each edit group, VuRLE identifies pairs of edit blocks that are adjacent nodes in the edit graph. For each of these *edit pairs*, it creates a repair template. A repair template consists of an edit pattern, which specifies a sequence of transformative edits, and a context pattern, which specifies where the edits should be applied.

To create an edit pattern from a pair of edit blocks, VuRLE compares the edit operation sequences of the two edit blocks. It then extracts the longest common substring (LCS) from the two sequences. This LCS is the edit pattern.

To create a context pattern from a pair of edit blocks, VuRLE processes the context of each edit block. Each context is a subtree. Given a pair of edit block contexts (which is a pair of AST subtrees, ST_1 and ST_2), VuRLE proceeds in the following steps:

1. VuRLE performs pre-order traversal on ST_1 and ST_2.

2. For each subtree, it extracts an ordered set of paths from the root of the subtree to each of its leaf nodes. The two ordered sets PS_1 and PS_2 represent the

context of ST_1 and ST_2 respectively. We refer to each of these paths as a *concrete* context sequence.

3. VuRLE then compares the corresponding elements of PS_1 and PS_2. For each pair of paths, if they share a longest common substring (LCS) of size T_{Sim}, we use this LCS to represent both pairs and delete the paths from PS_1 and PS_2. We refer to this LCS as an *abstract* context sequence.

4. VuRLE uses the remaining concrete sequences and identified abstract sequences as the context pattern.

As a final step, for each template, VuRLE replaces all concrete identifier types and names with placeholders. All occurrences of the same identifier type or name will be replaced by the same placeholder.

Figure 4.4 illustrates how VuRLE generates a context pattern by comparing two contexts. VuRLE performs pre-order traversal on AST subtrees of context 1 and context 2, generating an ordered set of paths for each context. After comparing the two set, VuRLE finds the matching paths highlighted in gray. For each matching pair of nodes that is of type SimpleName or SimpleType, VuRLE creates placeholders for it. There are five matching pair of nodes fulfilling this criteria, which are indicated by the dashed lines. Thus, VuRLE creates five placeholders named $\$V_0$, $\$V_1$, $\$V_2$, $\$T_0$, and $\$M_0$ from them.

4.4 Repair Phase: Repairing Vulnerable Applications

In this phase, VuRLE uses repair templates generated in the learning phase to detect whether an input code is vulnerable and simultaneously applies appropriate edits to repair the vulnerability. The two steps involved in this phase (Edit Group Selection and Template Selection) are presented in more details below. They are

performed iteratively until VuRLE can no longer detect any vulnerability.

4.4.1 Edit Group Selection

To detect whether an input code is vulnerable, VuRLE needs to find the edit group with the highest *matching score*. VuRLE compares the input code (IC) with each edit group (EG) and computes the matching score as follows:

$$S_{matching}(IC, EG) = \sum_{T \in templates(EG)} S_{matching}(IC, T) \tag{4.3}$$

In the above equation, $templates(EG)$ is the set of templates corresponding to edit group EG, and $S_{matching}(IC, T)$ is the matching score between template T and IC. VuRLE computes the matching score between the template T and input code IC as follows:

1. VuRLE first generates an AST of the input code.

2. VuRLE performs pre-order traversal on this AST to produce an ordered set of paths. Each path is a sequence of AST nodes from the root of the AST to one of its leaf node. Let us denote this as IP.

3. VuRLE compares IP with the context of template T. If sequences in T can be matched with sequences in IP, the number of matching nodes is returned as a matching score. Abstract sequences need to be fully matched, while concrete sequences only need to be partially matched. Otherwise, the matching score is 0.

4.4.2 Template Selection

In the most matched edit group EG, there are likely to be multiple corresponding templates (i.e., $templates(EG)$ has more than one member). In this final step, we need to pick the most suitable template.

To find a suitable template, VuRLE orders templates in a descending order according to their matching scores and tries to apply templates in $templates(EG)$ one-by-one. To apply a template, VuRLE first finds a code segment whose context matches with the context of the template. It then replaces all placeholders in the template with concrete variable names and types that appear in the context of the code segment. Next, VuRLE applies each transformative edits specified in the edit operation sequence of the template to the code segment.

If the application of a template results in redundant code, VuRLE proceeds to try the next template. The template selection step ends when one of the templates can be applied without creating redundant code. The code segment where the template is applied to is marked as being vulnerable and the resultant code after the transformative edits in the template is applied is the corresponding repaired code.

4.5 Evaluation

This section evaluates the performance of VuRLE by answering two questions below:

RQ1 (Vulnerability Detection) How effective is VuRLE in detecting whether a code is vulnerable?

RQ2 (Vulnerability Repair) How effective is VuRLE in repairing the detected vulnerable codes? Why some vulnerable codes cannot be repaired by VuRLE?

The following sections first describe the settings of our experiments, followed by the results of the experiments which answer the above two questions.

Table 4.1: Types of Vulnerabilities in Our Dataset

Vulnerability Type	Description
Unreleased Resource	Failing to release a resource [52] before reusing it. It increases a system's susceptibility to Denial of Service (DoS) attack.
Cryptographic Vulnerability	Inappropriate usage of encryption algorithm [18, 51] or usage of Plaintext Password Storage. It increases a system's susceptibility to Chosen-Plaintext Attack (CPA), brute force attack, etc.
Unchecked Return Value	Ignoring a method's return value. It may cause an unexpected state and program logic, and possibly a privilege escalation bug.
Improper Error Handling	Showing an inappropriate error handling message. It may cause a privacy leakage, which reveals useful information to potential attackers.
SSL Vulnerability	Unchecked hostnames or certificates [25, 20]. It makes a system susceptible to eavesdroppings and Man-In-The-Middle attacks.
SQL Injection Vulnerability	Unchecked input of SQL. It makes a system susceptible to SQL injection attack, which allows attackers to inject or execute SQL command via the input data [49].

4.5.1 Experiment Setup

Dataset. We collect 48 applications written in Java from GitHub[1] that have more than 400 stars. These applications consist of Android, web, word-processing and multimedia applications. The size of Android applications range from 3-70 MB while the size of other applications are about 200 MB. Among these applications, we identify vulnerabilities that affects them by manually analyzing commits from each application's repository. In total, we find 279 vulnerabilities. These vulnerabilities belong to several vulnerable types listed in Table 4.1.

Experiment Design. We use 10-fold cross validation to evaluate the performance of VuRLE. First, we split the data into 10 groups (each containing roughly 28 vul-

[1]Github: https://github.com/

Table 4.2: Detection Result: VuRLE vs LASE

	# of Detected Vulnerabilities	Precision	Recall
VuRLE	183	64.67%	65.59%
LASE	58	52.73%	20.79%

nerabilities). Then, one group is defined as a test group, and the other 9 groups as a training group. The test group is the input of the repair phase, while the training group is the input of the learning phase. We repeat the process 10 times by considering different group as test group. We examine the repaired code manually by comparing it with the real repaired code provided by developers. Furthermore, we compare VuRLE with LASE [53], which is state-of-the-art tool for learning repair templates. When running VuRLE, by default we set T_{Sim} to three.

To evaluate the vulnerability detection performance of our approach, we use precision and recall as the evaluation metrics, which are defined as follows.

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

where TP is the number of correctly detected vulnerabilities, FP is the number of wrongly detected vulnerabilities, and FN is the number of vulnerabilities that are not detected by our approach.

To evaluate the vulnerability detection performance of our approach, we use success rate as the evaluation metric. Success rate is the proportion of the correctly detected vulnerabilities that can be successfully repaired.

4.5.2 RQ1: Vulnerability Detection

To answer this RQ, we count the number of vulnerabilities that can be detected by VuRLE and compute the precision and recall on the entire dataset.

Table 4.2 shows the number of detected vulnerabilities, precision, and recall of

Table 4.3: Vulnerability Repair: VuRLE & LASE

	# of Repaired Vulnerabilities	Success Rate
VuRLE	101	55.19%
LASE	21	36.21%

VuRLE and LASE. VuRLE successfully detects 194 vulnerabilities out of the 279 vulnerabilities, achieving a recall of 65.59%. On the other hand, LASE can only detect 58 vulnerabilities out of the 279 vulnerabilities, achieving a recall of only 20.79%. Thus, VuRLE detects 215.52% more vulnerabilities compared to LASE. In terms of precision, VuRLE improves over LASE by 22.64%. It means that VuRLE proportionally generates less false positives than LASE.

4.5.3 RQ2: Vulnerability Repair

To answer this RQ, we investigate the number of vulnerabilities that can be repaired successfully. We also investigate how VuRLE can repair some bugs than cannot be repaired by LASE. We also discuss some causes on why VuRLE cannot repair some bugs.

Table 4.3 presents the success rate of VuRLE and LASE. The success rate of VuRLE is much higher than the success rate of LASE. VuRLE successfully repairs 101 vulnerabilities (55.19%), and LASE can only repairs 21 vulnerabilities, with a success rate of 36.21%. Thus, VuRLE can repair 380.95% more vulnerabilities compared to LASE. In terms of success rate, it improves over LASE by 52.42%.

Figure 4.5 provides a repair example generated by LASE and VuRLE on the same input code. The piece of code in the example contains a vulnerability that allows any hostname to be valid. LASE generates an overly general repair template, which only include invocation to `setHostnameVerifier`. It generate such template since each repair example invokes the `setHostNameVerifier` method after they define the `setDefaultHostnameVerifier` method, but the definition of the verifier method itself is different. On the other hand, VuRLE

generates two repair templates that can repair this vulnerability. One of the template is for modifying the `verify` method, and another is for invoking the `setDefaultHostnameVerifier` method.

Among 183 detected vulnerabilities, VuRLE cannot repair some of them. We discuss the main causes as follows:

Unsuccessful Placeholder Resolution. When replacing placeholders with concrete identifier names and types, VuRLE may use a wrong type or name to fill the placeholders. For example, the required concrete type is "double", but the inferred concrete type is "int". Moreover, VuRLE may not be able to concretize some placeholders since they are not found in the matching context.

Lack of Repair Examples. In our dataset, some vulnerabilities, such as *Cryptographic Misuses* and *Unchecked Return Value*, have many examples. Thus, a more comprehensive set of repair templates can be generated for these kinds of vulnerabilities. However, some vulnerabilities, such as *SSL Socket Vulnerability*, only have a few examples. Thus, VuRLE is unable to derive a comprehensive set of repair template to repair these kinds of vulnerabilities.

Partial Repair. For some cases, VuRLE can only generate a partial repair. This may be caused either by the inexistence of similar repairs or because VuRLE only extracts a partial repair pattern.

4.6 Conclusion and Future Work

In summary, we propose a tool, called VuRLE, to automatically detect and repair vulnerabilities. It does so by learning repair templates from known repair examples and applying the templates to an input code. Given repair examples, VuRLE extracts edit blocks and groups similar edit blocks into an edit group. Several repair templates are then learned from each edit group. To detect and repair vulnerabilities, VuRLE finds the edit group that matches the most with the input code. In

this group, it applies repair templates in order of their matching score until it detects no redundant code (in which case a vulnerability is detected and repaired) or until it has applied all repair templates in the edit group (in which case no vulnerability is detected). VuRLE repeats this detection and repair process until no more vulnerabilities are detected.

We have experimented on 48 applications with 279 real-world vulnerabilities and performed 10-fold cross validation to evaluate VuRLE. On average, VuRLE can automatically detect 183 (65.59%) vulnerabilities and repair 101 (55.19%) of them. On the other hand, the state-of-the-art approach named LASE can only detect 58 (20.79%) vulnerabilities and repair 21 (36.21%) of them. Thus, VuRLE can detect and repair 215.52% and 380.95% more vulnerabilities compared to LASE, respectively.

In the future, we plan to evaluate VuRLE using more vulnerabilities and applications written in various programming languages. We also plan to boost the effectiveness of VuRLE further so that it can detect and repair more vulnerabilities.

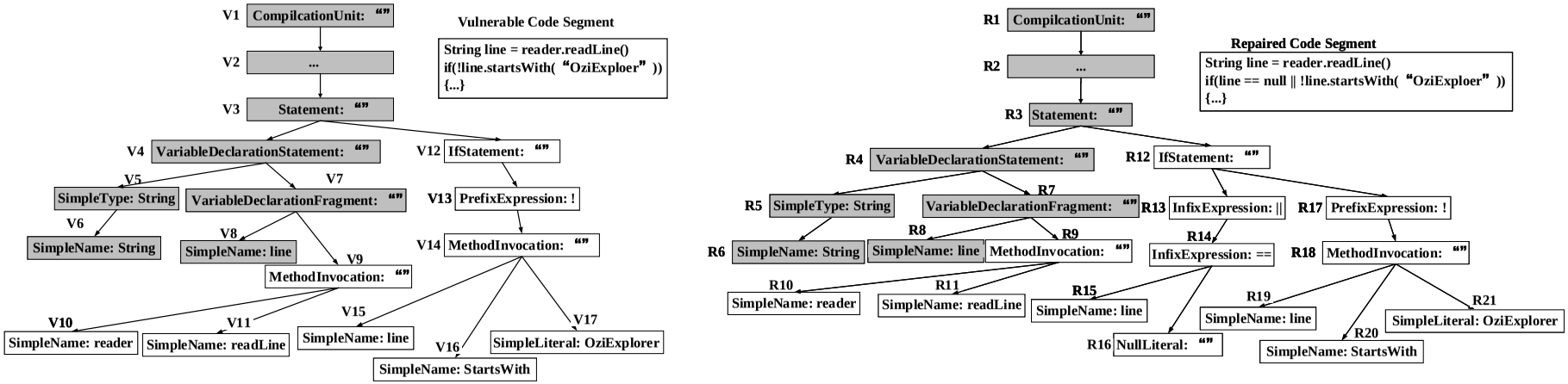


Figure 4.2: Vulnerable and Repaired Code Segments and Their ASTs

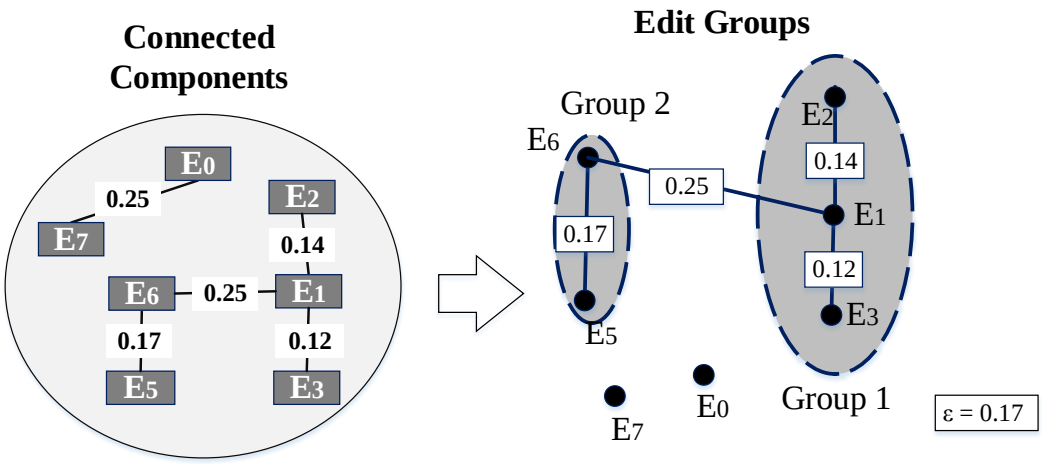


Figure 4.3: Edit Block Clustering: CCs to Edit Block Groups

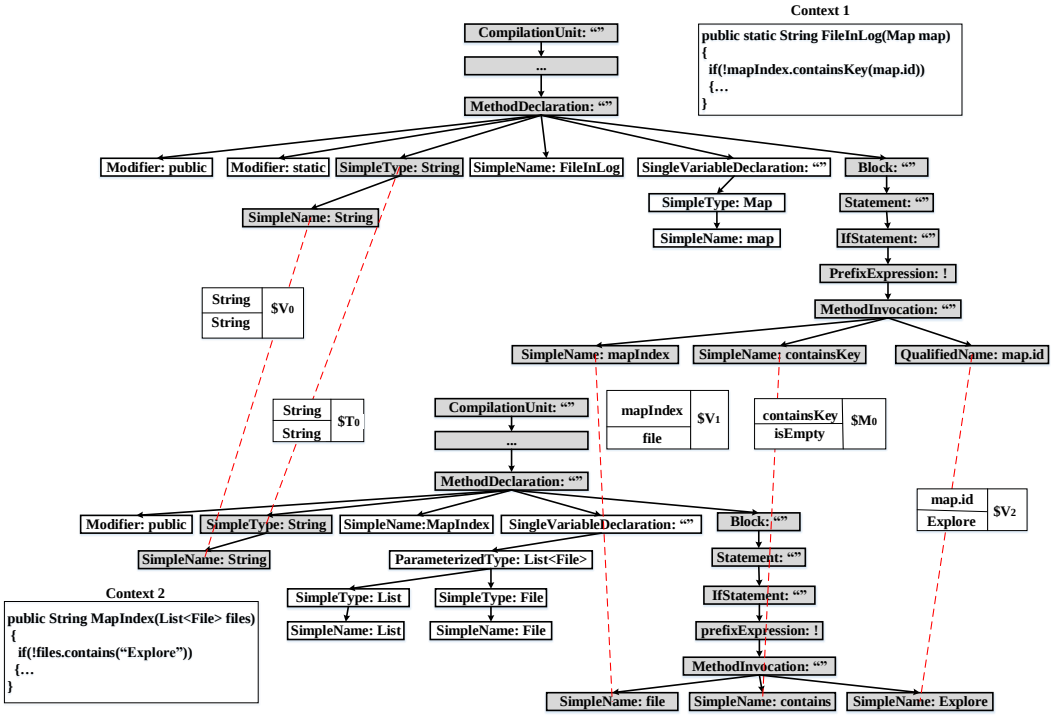


Figure 4.4: Context Pattern Generation

```
    HostnameVerifier allHostsValid = new HostnameVerifier(){
        public Boolean verify(String hostname, SSLSession session){
            return true;
        }
    }
-   urlConnection.setDefaultHostnameVerifier(allHostsValid);
+   urlConnection.setHostnameVerifier(allHostsValid);
```

(a) Patch Generated by LASE

```
    HostnameVerifier allHostsValid = new HostnameVerifier(){
        public Boolean verify(String hostname, SSLSession session){
-           return true;
+           HostnameVerifier hv = HttpsURLConnection.getDefaultHostnameVerifier();
+           Return hv.verify(hostname, session);
        }
    }
-   urlConnection.setDefaultHostnameVerifier(allHostsValid);
+   urlConnection.setHostnameVerifier(allHostsValid);
```

(b) Patch Generated by VuRLE

Figure 4.5: A Vulnerability Repaired by LASE and VuRLE

Chapter 5

Future Research Direction: An Empirical Study of Authentication Misuses in Android Applications

5.1 Introduction

Vulnerabilities introduced in previous two chapters only exist in the user side. However, there are some vulnerabilities that can only be detected during data transmission, such as communication between client and server. This chapter introduces our future research direction to detect authentication misuse flaws in Android applications. Based on the report published by OWASP [1] in 2016 and 2017, insecure authentication has been the top-10 vulnerabilities in applications.

As the number of smartphones have been rapidly increasing used nowadays. A smartphone has become a tool with multiple functions, such as socialising with others, working, online shopping, by applying different kinds of applications. Most web applications on smartphone []provide a login system, which requests for user's basic information(e.g., username, password, email, etc.). From user's perspective, those basic information are provided to verify their identity. To establish a secure communication channel, those basic information should be preprocessed be-

fore transmission, in case of the man-in-the-middle attack. A security expert knows the correct way to implement a secure "Challenge-Response" authentication. Most developers of web applications are usually not security experts. Due to the limited time and security knowledge, the "Challenge-Response" authentication may not be implemented correctly in those web applications. Three types of authentication protocols that are mainly used in Android applications: **authentication protocol with shared secret key**, **authentication protocol with timestamp**, and **authentication protocol with public key**.

For the login system in a web application, most recent authentication vulnerability detection approaches focus on input validation vulnerabilities [59, 17], which causes cross-site scripting(XSS) attack and SQL injection attack. Some approaches focus on password authentication vulnerabilities [75, 35] that are vulnerable to offline dictionary attack and impersonate attack. XSS vulnerability and SQL injection vulnerability are caused by improper input sanitization, which requires validation of an external input. Logic authentication vulnerability [22, 63] is caused by improper input assignment, such as authentication backdoor. It is related to internal value assignment(i.e., an input generated by developers). Firmalice [63] applies control flow analysis to detect a logic vulnerability(i.e., authentication backdoor). However, the connection between a client and a server may also be vulnerable to eavesdropping, intercepting, or manipulating while authenticating. For example, the message for "Challenge-Response scheme can not only be assigned by the data from an internal input, but also an internal input. We focus on two categories of authentication protocol vulnerability, authentication logic vulnerability and request forgery vulnerability.

The reason why we apply the detection on Android platform is described as follows: First, there are lots of third party application stores for users to download Android applications. Due to the existence of the large amount of third party, uploaded applications are not been checked carefully, that is, more vulnerable applications and malicious applications are uploaded. Second, our tool is written in Java,

and Android is closely related to Java. Also, Google provides lots of authentication related APIs. For example, the API `GoogleSignInAccount.getIdToken`, which extracts user's unique idToken that will be provided to server to authenticate user's identity. Third, Android is a open-source platform, and lots of open-sourced tools are published to decompile an apk file and translate the bytecode into an intermediate language, such as Soot [73], APKtool [83].

Our future research focus on detecting the incorrect implementation of authentication protocols (i.e., misuses of "Challenge-Response" authentication scheme). We detect the three protocols that are mainly used in Android applications, that is, **authentication protocol with shared secret key**, **authentication protocol with timestamp**, and **authentication protocol with public key**. By applying static program analysis, we are able to extract the dataflow of a challenge or a response to detect whether the "Challenge-ReSsponse" scheme is implemented correctly.

5.2 Definition of Authentication Protocols

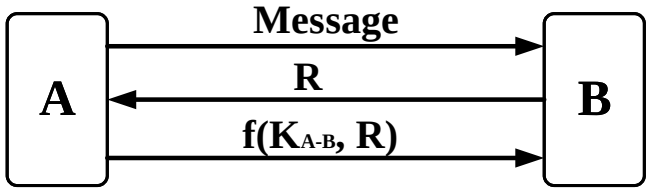
This section explains common protocols of login. Generally, an authentication contains two steps:

1. A sends a message, which includes her name and password, across the network to B.

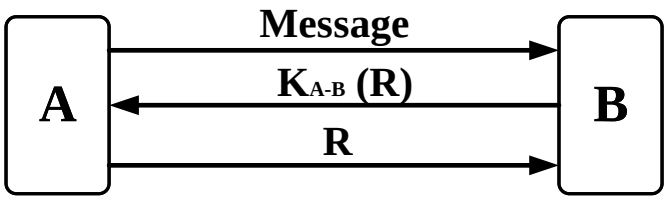
2. B verifies message (i.e., name and password) and starts the communication if it is matched.

However, the exchange data between A and B is not encrypted by any cryptographic integrity protection method. Three protocols that are commonly used in an authentication scheme, is described below: **protocol with shared secret key**, **protocol with timestamp** and **protocol with one-way public key**.

Protocol with Shared Secret Key Figure 5.1 describes two "Challenge-Response" protocols based on shared secret key. By using the secret



(a) Encrypted by A



(b) Encrypted by B

Figure 5.1: Login Authentication Protocol with Shared Secret Key

key K_{A-B} shared by A and B, challenge R can be encrypted through a symmetric encryption scheme (e.g., DES and AES) or be hashed into a message digest as a result. In Figure 5.1a, A encrypts the challenge R from B by using the secret key K_{A-B} as $f_{K_{A-B},R}$. To verify A, B uses K_{A-B} to decrypt ciphertext and extracts the challenge R. Another authentication protocol in Figure 5.1b, B sends an encrypted challenge $f_{K_{A-B},R}$ to A. A sends the decrypted challenge R back to B. Moreover, the challenge R and K_{A-B} can be concatenated. The result can be hashed as $hash(K_{A-B}, R)$.

Protocol with Timestamp In order to create a challenge R with limited lifetime, an authentication protocol with timestamp is proposed, shown in Figure 5.2. In this protocol, it requires that A and B have a synchronized clock, and A sends a message with an encrypted current time. Then, B extracts the time to ensure that it is validate. Moreover, this protocol is more efficient by reducing the authentication to a one-round protocol.

Protocol with One-Way Public Key Figure 5.3 illustrates two authentication protocols. In Figure 5.3a, A uses her private key to sign challenge R as $[R]_A$. B

can verify A by using her public key. If R matches, verification is succeed. Furthermore, B can use A's public key to encrypt the challenge R as $\{R\}_A$, shown in Figure 5.3b. A extracts R by using her private key to decrypt the ciphertext.

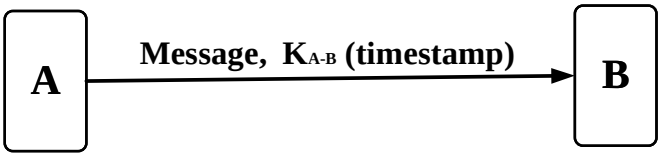


Figure 5.2: Login Authentication Protocol with TimeStamp

5.3 Common Rules of Password Authentication in Android

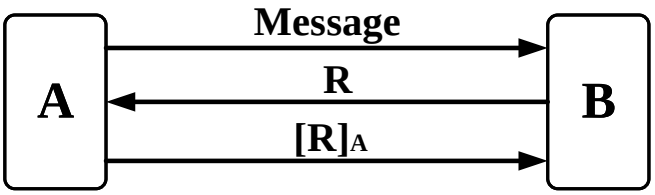
While the secure authentication protocol is precisely defined in Section 5.2, we propose the question whether developers who use authentication protocols implement the authentication correctly. Using authentication protocols correctly can be challenging. Several rules are defined as follows to implement various authentication protocols. In particularly, any application that violates one of the following rules will not be secure.

Rules for Challenge-Response Authentication Protocols. Password authentication is the simplest "Challenge-Response" authentication protocol, We have defined three general rules as follows:

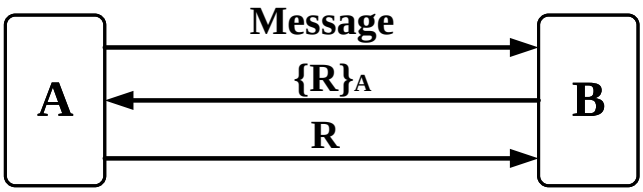
Rule 1: Do not use password in plaintext to transit in the public channel without any additional protection, such as SSL/TLS.

Rule 2: Do not use repeatable number as challenge.

Rule 3: Do not use predictable number in plaintext as a challenge if it is used alone to extract a session key.



(a) Use public key signature



(b) Use public key to encrypt

Figure 5.3: Login Authentication Protocol with Public Key

Rule 1 forbids to use password in plaintext if a public channel is not protected by any additional protection, such as SSL or TLS. The authentication communication is vulnerable to password stealing and man-in-the-middle attack (MITMA) [70]. Password stealing is an effective method to achieve attacker's goal. By launch phishing attack, attacker can extract user's password easily without decrypt it. Moreover, attacker is able to launch password reuse attack by using both username and password. Another threat is man-in-the-middle attack (MITMA). Suppose that an attacker has fully controlled the data exchange channel (i.e., eavesdropping, interception, and manipulation) [16], the attacker enables to eavesdrop the user's information (i.e., username and password), even modify user's information.

Rule 2 states that an authentication protocol should not use a repeatable number as challenge. For each "Challenge-Response" sequence, authentication protocols usually employ a unique cryptographic nonce as the challenge. It prevents the authentication against man-in-the-middle attack and subsequent replay attack.

Rule 3 states that the challenge should be unpredictable and encrypted, if it will be used to generate a session key. Since the challenge will be used to combine with the secret to generate an unpredictable encryption key for this session [12], an

attacker is easier to extract session key and decrypt all exchanged messages.

Rule for Timestamp-based Protocol. Each authentication is issued a timestamp to ensure that each authentication is unique. We define a rule for timestamp-based protocol.

Rule 4: Do not use a repeatable timestamp.

Rule 4 states that a timestamp for each authentication should be unique. Instead of applying a three-round authentication (i.e., "Challenge-Response" authentication protocol), the authentication with timestamp is more efficient to apply one-round authentication. However, if a timestamp in form of *hour/minute/second*, which is a repeatable timestamp that can be repeated on the next day with the same time, the attacker is able to impersonate the user to communicate with server.

Chapter 6

Dissertation Summary and Future Work

6.1 Summary of Contribution

This dissertation makes contributions on learning fix patterns of vulnerabilities and repair vulnerabilities of designing automatic vulnerability repair schemes. Moreover, our future research direction are described.

Our first work introduced an automatic vulnerability repair tool. We focused on cryptographic misuse defects, since cryptography is widely applied to protect user's data, especially on mobile platform. We introduced seven cryptographic algorithm that are commonly used in Android applications, and then manually created repair templates by analyzing a correct implementation for each cryptographic algorithm. To repair a vulnerable Android application, we detected a misuse and identify the misuse type by comparing the vulnerable code with every cryptographic misuse template. The corresponding repair template was customized and applied to repair the misuse by replacing variable names. Our result shows that our repair scheme is lightweight to be applied to Android apps and it is able to repair those vulnerabilities effectively.

In the second work, we made an attempt to learn repair edits automatically in-

stead of generating manually. We used known repair examples, including vulnerable code and repaired code, to learn repair edits of different vulnerabilities. Similar repair edits were clustered into an edit group, and several templates were generated from each edit group. A template was applied to the most matched vulnerability. Our experiment result further showed that it is possible to learn repair edits and repair multiple vulnerabilities automatically. This work has been published to European Symposium on Research in Computer and Science

Finally, we proposed our future research direction that we plan to detect authentication misuses on mobile platform. By analyzing the web application rules on mobile platform, we identified several vulnerabilities that are caused by the incorrect implementation of "Challenge-Response" authentication protocol. In mobile platform, three authentication protocols are commonly used, and we summarized six authentication secure rules to help developers implement a correct authentication protocol.

6.2 Future Work

Designing a more effective and applicable vulnerability repair scheme is a significant work to help developers and users to prevent to be attacked. In order to form a more comprehensive dissertation work, we are going to cover more studies in the final dissertation. In this section, we introduce the studies that we will do in the future and present a concrete plan to finish them and the dissertation.

6.2.1 Future Work: Unknown Vulnerabilities Detection and Repair

Authentication Misuses Detection and Repair

As authentication protocols are widely used nowadays, designing a usable and light-weight tool to detect authentication misuses is the extremely important. Lots

of secure authentication protocols are proposed for various of web services [94, 31], but only few of them can be applied on mobile platforms because of the resources and memory limitation. To implement a light-weight authentication protocol, developers have to follow the secure authentication implementation strictly. However, most of them are not security experts that some implementations are incorrect. We plan to propose a light-weight tool to detect the misuses of authentication protocols on mobile platform. This tool is required to detect precisely and effectively.

Unknown Vulnerabilities Detection and Repair

Most previous works deal with known vulnerabilities detection and repair. We also propose some approaches to repair some known vulnerabilities automatically. However, unknown vulnerabilities are more dangerous that attackers can exploit it and perform new attacks on an unknown vulnerability to steal users' private information. ShieldGen [81] detects and repairs unknown vulnerabilities. It uses zero-day attack to identify those unknown vulnerabilities if they are vulnerable to those attacks. However, the data patch generation is only performed on the input data. By comparing an input with legitimate information, it is able to identify the incorrect input and correct it. Since only few vulnerabilities that can be exploited by using malicious inputs, patch malicious input only repairs specific vulnerabilties. We aim to generate a model with several safe behaviors. We expected to use the model to compare an input application those benign behaviors. If the input application has different behaviors, we could assume this application is vulnerable. Next, we can repair its detected vulnerable behaviors to safe behaviors.

Bibliography

[1] Owasp mobile top 10. *URL: https://www.owasp.org/index.php/Mobile-Top-10-2016*.

[2] Securerandom. *URL: http://developer.android.com/reference/java/security/SecureRandom.html*.

[3] Seven templates for the corresponding misuses. *URL: https://sites.google.com/a/smu.edu.sg/my-work/home*.

[4] Status of software security report, vol 5. *URL: http://www.veracode.com/resources/state-of-software-security*, 2013.

[5] A. Alavi, A. Quach, H. Zhang, B. Marsh, F. U. Haq, Z. Qian, L. Lu, and R. Gupta. Where is the weakest link? a study on security discrepancies between android apps and their website counterparts. In *International Conference on Passive and Active Network Measurement*, pages 100–112. Springer, 2017.

[6] M. A. Alkhalaf. *Automatic Detection and Repair of Input Validation and Sanitization Bugs*. University of California, Santa Barbara, 2014.

[7] F. Amiri, M. R. Yousefi, C. Lucas, A. Shakery, and N. Yazdani. Mutual information-based feature selection for intrusion detection systems. *Journal of Network and Computer Applications*, 34(4):1184–1199, 2011.

[8] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan. Automated testing for sql injection vulnerabilities: an input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 259–269. ACM, 2014.

[9] A. Armando, A. Merlo, M. Migliardi, and L. Verderame. Breaking and fixing the android launching flow. *Computers & Security*, 39:104–115, 2013.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[11] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan. Candid: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 12–24. ACM, 2007.

[12] M. Bellare and P. Rogaway. Provably secure session key distribution: the three party case. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 57–66. ACM, 1995.

[13] L. Bilge and T. Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844. ACM, 2012.

[14] A. M. Braga and D. C. Schwab. Design issues in the construction of a cryptographically secure instant message service for android smartphones. *SECURWARE 2014*, page 18, 2014.

[15] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[16] F. Callegati, W. Cerroni, and M. Ramilli. Man-in-the-middle attack to the https protocol. *IEEE Security & Privacy (sp)*, 7(1):78–81, 2009.

[17] C. Cao, N. Gao, P. Liu, and J. Xiang. Towards analyzing the input validation vulnerabilities associated with android system services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 361–370. ACM, 2015.

[18] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[19] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Knowledge Discovery and Data Mining (KDD)*, volume 96, pages 226–231, 1996.

[20] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[21] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[22] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX security symposium*, volume 58, 2010.

[23] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11), 2007.

[24] F. Gao, L. Wang, and X. Li. Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 786–791. IEEE, 2016.

[25] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[26] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[27] W. G. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183. ACM, 2005.

[28] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.

[29] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[30] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 66–77. ACM, 2014.

[31] S. Kalra and S. K. Sood. Secure authentication scheme for iot and cloud servers. *Pervasive and Mobile Computing*, 24:210–223, 2015.

[32] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[33] S. H. Kim, D. Han, and D. H. Lee. Predictability of android openssl's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications Security*, pages 659–668. ACM, 2013.

[34] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 61–72. ACM, 2016.

[35] W.-C. Ku and S.-T. Chang. Impersonation attack on a dynamic id-based remote user authentication scheme using smart cards. *IEICE Transactions on Communications*, 88(5):2165–2167, 2005.

[36] X.-B. D. Le. Towards efficient and effective automatic program repair. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 876–879. ACM, 2016.

[37] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604. ACM, 2017.

[38] X. B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 213–224. IEEE, 2016.

[39] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[40] I. Lee, S. Jeong, S. Yeo, and J. Moon. A novel method for sql injection attack detection based on removing sql query attribute values. *Mathematical and Computer Modelling*, 55(1):58–68, 2012.

[41] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, pages 1193–1204. ACM, 2013.

[42] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[43] L. Li, A. Bartel, J. Klein, and Y. Le Traon. Automatically exploiting potential component leaks in android applications. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 388–397. IEEE, 2014.

[44] Y. Li, Y. Zhang, J. Li, and D. Gu. icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In *Network and System Security*, pages 349–362. Springer, 2014.

[45] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv. Netshield: massive semantics-based vulnerability signature matching for high-speed networks. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 279–290. ACM, 2010.

[46] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie. Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, Computer & Communications Security*, pages 329–340. ACM, 2007.

[47] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 2–13. IEEE Press, 2017.

[48] J. Liu, T. Wu, J. Yan, and J. Zhang. Fixing resource leaks in android apps with light-weight static analysis and low-overhead instrumentation. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 342–352. IEEE, 2016.

[49] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 22nd USENIX security symposium*, volume 2013, 2005.

[50] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer & Communications Security*, pages 229–240. ACM, 2012.

[51] S. Ma, D. Lo, T. Li, and R. H. Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer & Communications Security*, pages 711–722. ACM, 2016.

[52] N. Meghanathan. Source code analysis to remove security vulnerabilities in java socket programs: A case study. *arXiv preprint arXiv:1302.1338*, 2013.

[53] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511. IEEE Press, 2013.

[54] M. Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[55] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX security symposium*, pages 543–558, 2013.

[56] G. Pellegrino and D. Balzarotti. Toward black-box detection of logic flaws in web applications. In *NDSS*, 2014.

[57] W. Qiang, Y. Liao, G. Sun, L. T. Yang, D. Zou, and H. Jin. Patch-related vulnerability detection based on symbolic execution. *IEEE Access*, 5:20777–20784, 2017.

[58] M. Salas and E. Martins. Security testing methodology for vulnerabilities detection of xss in web services and ws-security. *Electronic Notes in Theoretical Computer Science*, 302:133–154, 2014.

[59] T. Scholte, D. Balzarotti, and E. Kirda. Have things changed now? an empirical study on input validation vulnerabilities in web applications. *Computers & Security*, 31(3):344–356, 2012.

[60] E. C. Sezer, P. Ning, C. Kil, and J. Xu. Memsherlock: an automated debugger for unknown memory corruption vulnerabilities. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 562–572. ACM, 2007.

[61] L. K. Shar and H. B. K. Tan. Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10):1767–1780, 2013.

[62] L. K. Shar, H. B. K. Tan, and L. C. Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 642–651. IEEE Press, 2013.

[63] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[64] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*, pages 75–80. IEEE, 2014.

[65] S. Sidiroglou-Douskos, E. Lahtinen, and M. Rinard. Automatic discovery and patching of buffer and integer overflow errors. 2015.

[66] A. Smirnov and T.-c. Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *The Network and Distributed System Security Symposium*, 2005.

[67] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 305–316. IEEE, 2010.

[68] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *The Network and Distributed System Security Symposium*, 2013.

[69] P. A. Sonewar and N. A. Mhetre. A novel approach for detection of sql injection and cross site scripting attacks. In *Pervasive Computing (ICPC), 2015 International Conference on*, pages 1–4. IEEE, 2015.

[70] H.-M. Sun, Y.-H. Chen, and Y.-H. Lin. opass: A user authentication protocol resistant to password stealing and password reuse attacks. *IEEE Transactions on Information Forensics and Security*, 7(2):651–663, 2012.

[71] V. Sunkari and C. G. Rao. Preventing input type validation vulnerabilities using network based intrusion detection systems. In *Contemporary Computing and Informatics (IC3I), 2014 International Conference on*, pages 702–706. IEEE, 2014.

[72] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1232–1243. ACM, 2014.

[73] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[74] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *The Network and Distributed System Security Symposium*, volume 2007, page 12, 2007.

[75] D. Wang and P. Wang. Offline dictionary attack on password authentication schemes using smart cards. In *Information Security*, pages 221–237. Springer, 2015.

[76] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu. Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 61–70. ACM, 2015.

[77] L. Wang, S. Jajodia, A. Singhal, P. Cheng, and S. Noel. k-zero day safety: A network security metric for measuring the risk of unknown vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 11(1):30–44, 2014.

[78] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[79] X. Wang and H. Yu. How to break md5 and other hash functions. In *In proceedings of 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques(Advances in Cryptology–EUROCRYPT)*, pages 19–35. Springer, 2005.

[80] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[81] H. J. W. Weidong Cui, Marcus Peinado and M. E. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Symposium on Security and Privacy*. IEEE, 2007.

[82] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[83] R. Winsniewski. Apktool: a tool for reverse engineering android apk files. *URL: https://ibotpeaches. github. io/Apktool/(vi sited on 07/27/2016)*, 2012.

[84] Y. Wu, B. Chen, Z. Zhao, and Y. Cheng. Attack and countermeasure on interlock-based device pairing schemes. *IEEE Transactions on Information Forensics and Security*, 13(3):745–757, 2018.

[85] Q. Xin and S. P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 660–670. IEEE Press, 2017.

[86] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE, 2014.

[87] T. Yang, H. Cui, S. Niu, and P. Zhang. An analysis on sensitive data passive leakage in android applications. In *Communication Technology (ICCT), 2015 IEEE 16th International Conference on*, pages 125–131. IEEE, 2015.

[88] T. Ye, L. Zhang, L. Wang, and X. Li. An empirical study on detecting and fixing buffer overflow bugs. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 91–101. IEEE, 2016.

[89] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, 2014.

[90] F. Yu, C.-Y. Shueh, C.-H. Lin, Y.-F. Chen, B.-Y. Wang, and T. Bultan. Optimal sanitization synthesis for web application vulnerability repair. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 189–200. ACM, 2016.

[91] M. Zhang, L. Wang, S. Jajodia, A. Singhal, and M. Albanese. Network diversity: a security metric for evaluating the resilience of networks against zero-day attacks. *IEEE Transactions on Information Forensics and Security*, 11(5):1071–1086, 2016.

[92] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

[93] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu. Embroidery: Patching vulnerable binary code of fragmentized android devices. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 47–57. IEEE, 2017.

[94] R. Zhou, Y. Lai, Z. Liu, Y. Chen, X. Yao, and J. Gong. A security authentication protocol for trusted domains in an autonomous decentralized system. *International Journal of Distributed Sensor Networks*, 12(3):5327949, 2016.