UNIVERSITY OF CALIFORNIA
RIVERSIDE

Discerning Code Changes From a Security Perspective

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Yue Duan

June 2019

Dissertation Committee:

Dr. Heng Yin, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Rajiv Gupta
Dr. Chengyu Song

The Dissertation of Yue Duan is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

Before everything else, I would like to say that I am very grateful to my advisor Dr. Heng Yin, for without whose help, I would not have been here. His knowledge, enthusiasm, encouragement and continuous support are the key to my Ph.D study.

Besides, my sincere gratitude go to the rest of my PhD disseration committee: Prof. Nael Abu-Ghazaleh, Prof. Rajiv Gupta and Prof. Chengyu Song, for their insightful questions and comments.

I thank my fellow labmates: Mu, Aravind, Qian, Andrew, Xunchao, Rundong, Jinghan, Brian, Wei, Sina, Lian, Jie, Zhenxiao, Chengheng, Lei, Pan and Jianlei, for their help during my hard times, for all the sleepless nights before deadlines and for all the fun we had in the past five years.

I would like to say thank you to my parents Xiaochang Duan and Lin Wu. Their unconditional love and support let me overcome all the difficulties and make me who I am.

Most importantly, I would like to thank my wife Rui Li. This dissertation would not be possible without your love and support.

This dissertation includes previously published material entitled "Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation" published in the Network and Distributed System Security Symposium, February 2018.

To my parents and my wife.

ABSTRACT OF THE DISSERTATION

Discerning Code Changes From a Security Perspective

by

Yue Duan

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2019
Dr. Heng Yin, Chairperson

Programs are not immutable. In fact, most programs are under constant changes for security (e.g, vulnerability fix) and non-security (e.g., new features) reasons. These code changes, however, could expose great security challenges. Android packers, as a set of code transformation techniques, are gaining increasingly popularity among Android malware, rendering existing malware detection techniques obsolete. Despite the importance of this emerging trend (app packing), no comprehensive study has ever been conducted to help the community understand the status quo of Android packing and unpacking techniques.

Android third-party libraries (TPL) that can provide complementary functionalities and ease the app developments have become one of the major sources of Android security issues due to the pervasive outdatedness issue. Prior efforts have been made to understand and mitigate specific types of security issues in TPLs, but there exists no generic solution to solve the issues and keep them up-to-date.

Binary Code Differential Analysis, a.k.a, binary diffing, is a fundamental analysis capability that aims to quantitatively measure the similarity between two given binaries and

produce the fine-grained block level matching. It has enabled many critical security usages including patch analysis and malware analysis. Existing binary diffing techniques suffer from low accuracy, poor scalability, coarse granularity or require extensive labeled training data to function. A new technique is needed to accurately and efficiently perform binary diffing at a fine-grained basic block level.

This dissertation addresses these problems by presenting concepts, methods and techniques to perform generic Android packer analysis, automatically generate updates for outdated TPLs and propose an novel unsupervised deep neural network based program-wide code representation learning technique for binary diffing.

Firstly, an Android packing analysis framework called DroidUnpack is developed to reliably capture and analyze unpacking behaviors on Android. It monitors the execution at the lowest level and reconstruct Java level semantics. In this way, it can catch the intrinsic "write-and-then-execute" unpacking behaviors at either native level or Java level or both. We further conduct the first systematic large-scale study on Android (Un)packers and report some surprising findings.

Secondly, LibBandAid is proposed to automatically generate updates for TPLs in Android apps in a non-intrusive fashion without the need of source code. It extracts the outdated libraries from apps and compare them to their latest versions. Then it analyzes the code changes and further performs updating in a way that it does not require any code modification on the app side and more importantly, introduce no impact to the library interactions with other components locally (e.g., underlying Android system) and remotely (e.g., server) as we call it *non-intrusive*.

Thirdly, DeepBinDiff is presented as an unsupervised deep neural network based program-wide code representation learning technique for binary diffing. It relies on both the code semantic information as well as the program-wide control flow information to generate block embeddings and further performs a K-hop greedy matching to find the optimal diffing results using the generated block embeddings.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer programs are not static. In fact, most of them are constantly being updated due to a variety of reasons including vulnerability fix, new feature implementation and intellectual property protection. These code changes, however, oftentimes expose great challenges when performing security analyses for both traditional binary programs as well as emerging mobile apps. In particular, we identify three important security problems to be our main focus: 1) packed Android malware, 2) outdated third-party libraries (TPLs) in Android apps and 3) binary diffing problem. These problems are critical to understanding and improving security and yet have not been addressed by existing techniques due to the challenges introduced by code changes.

Packed Android malware Studies [146, 135] show that both malicious and benign apps utilize packing techniques to hide their code. The complexity in analyzing obfuscated code, as introduced by these techniques, has become a new barrier to protecting Android users. Particularly, without in-depth understanding of Android packers, malicious and plagiarized

apps could easily circumvent the vetting process put in place by app markets and spread across Android devices through these markets [2, 11].

Despite the importance of this emerging trend (app packing), no comprehensive study, however, has ever been conducted to help the community understand the status quo of Android packing and unpacking techniques, which is crucial to building practical defense and mitigating the security risks brought in by these techniques. Moreover, this task cannot be done by any existing Android unpackers due to their design limitations. Concretely, current Android unpackers could be roughly categorized into three types based on distinct system designs. 1) signature-based unpackers (e.g., kisskiss [120]) locate DEX file by signature and perform memory dump; 2) DVM hooking-based unpackers (e.g., dexhunter [146]) modify DVM to hook certain important functions to find DEX file and then dump the code; 3) DVM data structure based unpackers (e.g., AppSpear [135] modify DVM to dump Dalvik data structures on the air and then assemble them back into a DEX file. None of them can handle *unknown* packing operations nor can they have any view for behaviors at native level. On the other hand, existing PC unpackers (e.g., Renovo [79]) are just designed for binary code and cannot handle Java code.

Outdated TPLs in Android apps Security vulnerabilities are thwarting the security of Android apps. More interestingly, the outdatedness problem of Third-party libraries (TPL) have become a new hot source for vulnerabilities [42, 58] as TPLs are constantly getting more popularity. It has been revealed [42] that 70.40% of Android apps include at least one outdated TPL and 77% of app developers update at most a strict subset of their included libraries since they do not have enough incentive to keep the TPLs up-to-date, leaving many

known security vulnerabilities unpatched within their apps. In fact, updating TPLs can be a non-trivial task for app developers by virtue of two major reasons. First, more than half (51.8%) of the libraries require code modification to the apps when updated to the latest versions due to library API changes [58]. In other words, updating libraries to the latest version is very likely to involve app modification. Second, although 97.8% of actively used library versions with a known security vulnerability could be fixed via a drop-in replacement with a fixed version [58], it is practically infeasible for app developers to manually find a suitable version with security fix to replace the vulnerable version without app modification for each and every library in their apps.

Prior efforts are made to study and mitigate problems with TPLs in Android apps. To understand TPLs in Android, variety of library detection techniques are proposed [96, 52, 42, 87, 88, 58, 125, 145] to detect TPLs in apps and perform measurement study to comprehend the library prevalence [87, 125, 145], library evolution [88], up-to-dateness [58] and other security related information [42, 58]. To further mitigate security problems with TPLs in Android apps, series of techniques are proposed to isolate TPLs from the Android app. TPLs can be transformed into new processes [116, 142], new apps [121, 78], or new services [106]. Other works enforce in-app privilege separations [127, 114] so as to keep the apps' privileges from TPLs. However, these techniques do not fix security issues per se but merely limit the harmfulness of potential problems in TPLs from the apps.

Android application patching techniques are a different set of research that could help with the TPL security issues. AppSealer [139] performs automatic patching for preventing component hijacking attacks in Android apps. Capper [140] and Liu et.al. [92] rewrite

the Android apps to keep track of private information flow and detect privacy leakage at runtime. CDRep [95] fixes cryptographic-misuses in Android with similar bytecode rewriting technique. Azim et.al. [41] detects crashes dynamically and uses bytecode rewriting technique to avoid such crashes in the future. Nonetheless, these techniques only aim to fix specific types of security issues and do not deal with the outdatedness problem on TPLs. Hence, existing patching techniques on Android cannot keep TPLs updated and fix security issues in a generic fashion.

Binary Diffing Binary Code Differential Analysis, a.k.a, binary diffing, is a fundamental analysis capability, which aims to quantitatively measure the similarity between two given binaries and produce the fine-grained block level matching. Given two input binaries, it precisely characterizes the program-wide differences by generating the optimal matching among the blocks with quantitative similarity scores. It can not only present a more precise, fine-grained and quantitative results about the differences at a whole binary scale but also explicitly reveal how code evolves across different versions or optimization levels. Due to this level of precision and granularity, it has enabled many critical security usages in different scenarios when program-wide analysis is required, such as changed parts locating [28], malware analysis [66, 101], patch analysis [132, 89], binary wide plagiarism detection [94] and patch-based exploit generation [40].

Because of the importance, binary diffing has been an active research focus. Existing techniques can be categorized into three categories: static, dynamic and learning-based approaches. Static approaches [34, 60, 108, 65, 57, 55, 56] usually perform many-to-many graph isomorphism detection on generated flow graphs [60, 108, 65] or decompose the bina-

ries into fragments [57, 55, 56] for similarity detection. These approaches do not consider the semantics of instructions which can be critical during analysis, especially when dealing with different compiler optimization levels. Moreover, traditional graph matching algorithm such as Hungarian algorithm [83] is expensive and cannot guarantee optimal matching.

Dynamic techniques, on the other hand, carry out the analysis by directly executing the given code [61, 126], performing dynamic slicing [100] or using symbolic execution [70, 99, 94] on the given binaries and checking the semantic level equivalence based on the information collected during the execution. In general, these techniques excel at extracting semantics of the code and have good resilience against compiler optimizations and code obfuscation but usually suffer from very poor scalability and incomplete code coverage because of the nature of dynamic analysis.

Recently, researchers have been leveraging the advance of machine learning to tackle the problem. Various techniques such as Genius [68], Gemini [131], INNEREYE [150] and Asm2Vec [59] have been proposed to utilize graph representation learning techniques [53, 98, 86] and incorporate code information into embeddings (i.e, high dimensional numerical vectors). Then they use these embeddings for similarity detection. However, existing learning-based techniques suffer from low accuracy, poor scalability, coarse granularity or require extensive labeled training data to function.

1.1 Thesis Statement

In a nutshell, the fundamental research question behind these security issues is how to understand the code changes and use the knowledge to solve security problems. And that

is the main focus of this thesis. Therefore, the thesis statement is **discerning program code changes from a security perspective has become essential to understand and further mitigate current security issues**.

To this end, we propose three novel techniques as illustrated in Figure 4.1. This thesis aims to tackle three interconnected security problems by leveraging these three techniques.

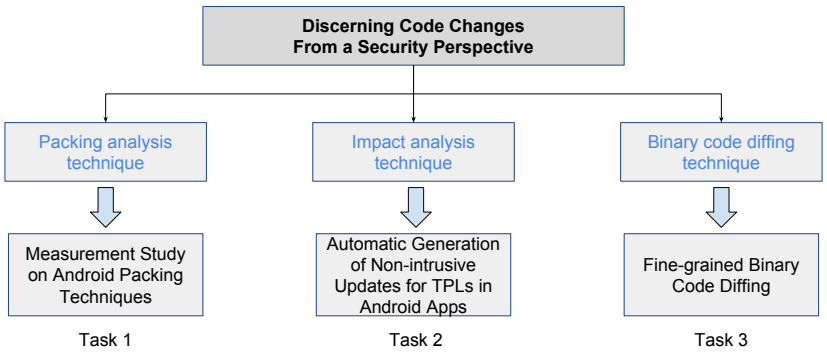


Figure 1.1: Overview of Thesis work

- **Packed Android Malware.** To battle the emerging packed Android malware, we propose a novel Android packing analysis technique called DroidUnpack based on a whole-system emulation to reliably capture and analyze unpacking behaviors. It monitors app execution at the lowest level, so we do not miss any behaviors related to unpacking. Then it reconstructs Java level execution, for accurate detection and better understanding of unpacking behaviors. Based on the technique, task 1 aims to comprehensively understand the current status of code changing technique (packing) that are used among malware to hide the malice from being analyzed. We conduct the first comprehensive study over 6 major commercial packers, 3 state-of-the-art unpackers and 93,910 Android malware samples in the wild and report the findings.

- **Outdated TPLs in Android apps.** With the goal of addressing the outdatedness issue of TPLs, we aim to automatically generate updates for TPLs in Android apps in a non-intrusive fashion without the need of source code. More specifically, we extract the code of outdated libraries from Android apps and compare them to their latest versions. Then we analyze the code changes and further perform updating in a way that it does not require any code modification on the app side and more importantly, introduce no impact to the library interactions with other components locally and remotely as we call it *non-intrusive*. To this end, we propose a novel technique called LibBandAid to generate patches that are guaranteed to be sound and concise.

- **Binary Diffing.** The last task is to solve the binary diffing problem. To overcome the limitations of existing techniques, we propose an unsupervised program-wide code representation learning technique named DeepBinDiff that relies on both the code semantic information as well as the program-wide control flow information to generate block embeddings. Furthermore, we propose a K-hop greedy matching algorithm to find the optimal diffing results using the generated block embeddings.

Chapter 2

Background

2.1 Android

Android has undoubtedly become the most popular mobile operating system as Google has announced over 2 billion monthly active Android devices and 2.8 million Android apps by 2017 [1]. Ever since the booming of Android, security problems have immediately followed and Android apps have turned into the breeding ground of various security issues. Security problems including malware [147], vulnerabilities (e.g., privilege escalation [54], permission re-delegation [67] and component hijacking [93]) and privacy leakage [64, 62, 76, 147, 149] have attracted much research focus. A series of techniques has been proposed to mitigate problems by detecting Android malware [133, 73, 35, 69, 138, 122], fixing vulnerabilities [139, 141] and preventing information leakage [63, 136]. Nevertheless, many of these security issues have become increasingly complex and render existing techniques obsolete when taking code changes into account.

2.1.1 Android System

Unlike traditional PC, Android system has a multi-level design. It is built on top of a customized Linux kernel. A process named Zygote is the parent for all Android app processes. Above the kernel, Android system provides a set of libraries including app runtime. The runtime coordinates apps with Android framework libraries so that the apps can interact with lower-level system through framework APIs. This fundamental design difference in Android system requires our tool to have multi-level views about the whole system including OS level, binary level and Java level views. Moreover, just like Android system, Android runtime environment is also very different from traditional PC, and has changed drastically over time.

2.1.2 Android Runtime Environment

Dalvik virtual machine. Legacy Android (version 4.4 and earlier) leverages Dalvik Virtual Machine (DVM) to interpret DEX bytecode programs at runtime. At install time, a `dexopt` tool optimizes the input DEX bytecode and creates `ODEX` files so as to improve runtime efficiency. Upon execution, DVM enables bytecode interpretation and translates DEX code to native code for the target architecture.

ART environment. The recent Android system (version 5.0 and later) has substituted Dalvik VM with the new Android Runtime (ART) in order to improve runtime performance. In contrast to the bytecode interpretation in DVM, ART conducts ahead-of-time (AOT) compilation to produce machine dependent code prior to execution. To do so, ART utilizes the compiler, `dex2oat`, to transform an input DEX executable into an OAT file. Internally,

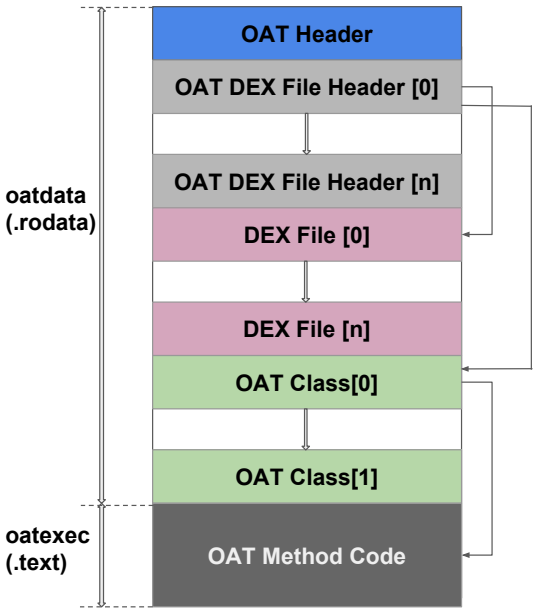


Figure 2.1: OAT file structure.

`dex2oat` can perform multiple rounds of optimizations and depending upon the existence of legacy code, it may select between "interpret" and "quick" modes to achieve different levels of optimizations. The "interpret" mode means no code will be compiled into native, while "quick" mode compiles as many codes as possible.

An OAT file is essentially an ELF dynamic object, but it follows a unique format. Figure 2.1 depicts the structure of an OAT file. At the beginning of the file resides the OAT header, which contains the count of enclosed DEX files. Subsequent to the header, there exists four major sections, OAT DEX file headers, original DEX files, OAT class headers and OAT methods. An OAT DEX file header holds offsets to the embedded DEX files and corresponding OAT class headers, respectively, while the latter further contains offsets to the compiled OAT methods. A DEX class can be completely, partially compiled or not compiled at all. Therefore, an OAT class header also keeps a bitmap to indicate whether a

method has been compiled. Only compiled methods are stored in an OAT file and referenced by the data structure in the OAT method section.

2.1.3 Android Apps

As a result, Android apps are designed to be quite different from traditional PC programs as well. Android apps are built as a combination of distinct components that can be invoked individually and can contain both Java and native parts. Native components are simply shared libraries that are dynamically loaded at runtime. The app runtime library (`libdvm.so` or `libart.so`) interprets or compiles the Java components to produce and launch native instructions. The Java Native Interface (JNI) is then used to enable communications between the native and Java sides. Thus, a packed Android app often packs its Java code as well as its native code (major program logic or critical functionality) into binary resource files. Usually, it still maintains a dummy Java component, which acts solely as a dispatcher to launch the unpacking procedure.

2.1.4 Unpacking techniques

Runtime packers in general have been well studied and series of solutions have been proposed to defeat them [111, 97, 79, 115]. However, due to the differences in so many aspects, there exists a major discrepancy in the unpacking techniques between Android and PC.

PC unpackers such as Omniunpack [97] and renovo [79] monitor and trace the packed program execution at native instruction or page granularity using either memory protection mechanism or emulated environment so that they can reliably uncover the pro-

gram behaviors at native level. Nonetheless, this kind of unpacking technique does not fit into Android scenario where apps contain both native and Java components. Fundamentally, the design lacks the capability of monitoring Java level behaviors, thus, will not be able to understand anything happens at that level.

On the other hand, existing Android unpackers [146, 135, 119] fall short of native side. Current Android unpackers can be roughly categorized into three types to extract code based on different design choices: 1) signature-based memory dump unpacker as Kisskiss [119]; 2) hooking-based memory dump unpacker as DexHunter [146]; 3) Dalvik data structures dumping and DEX file assembly unpacker as AppSpear [135]. All the Android unpackers rely on Java level information other than intrinsic nature of packed programs, which is, the original code will be dynamically generated and then executed [79]. As a result, none of them is able to detect and analyze *previously unknown* packing techniques and understand what happens at the native side let alone the interactions between Java and native.

2.1.5 Android Ecosystem

Last but not least, another significant and special characteristic about Android which has great influence on packing study is the unique Android ecosystem. This ecosystem applies a huge impact to both app developers and users. After the Android apps are developed, the developers upload them to Android app markets, e.g., Google Play. The market will perform necessary vetting process to the uploaded apps and make them available for the end users. By default, Android system disallows users to install apps outside of Google Play. Further, because of the vetting performed by those Android app markets, users tend

to trust them and download apps from them. According to 2016 Google I/O [6], Google Play has reached over 1 billion monthly active users in 2016 which makes it the world largest app distribution platform. As a result, malware and plagiarized apps that circumvent app markets security checks and infiltrate into this ecosystem can impose even bigger threat to users than normal malware. However, according to prior reports [7, 11], packing techniques can indeed help malicious developers sneak malware into Google Play. This fact motivates us to study Android packing techniques.

What's more, unlike traditional PC, commercial packing services have become a part of the Android ecosystem as well. They are being widely used by many developers to pack and protect their intellectual property before submitting to app markets [135]. In order to prevent people from abusing the services, they have enforced their own malware and plagiarism detection mechanism. Our study also would like to find out if these services can be exploited and abused by malicious users. And since all the Android commercial packing services are freely available to users, it gets us wondering about what their business models are and further motivates us to study the detailed behaviors of those packers.

2.2 Change Impact Analysis

Change Impact Analysis [46] or Impact Analysis for short studies how code changes in one place could affect codes in other places of the program. Many works have been proposed [85, 104, 110, 109, 37, 118, 82, 144, 112, 90] to improve the change impact analysis. Some of the works utilize call graph analysis to study the impact of code change [110, 109, 37]. The limitation for this kind of analysis is that call graph by nature can only provide a coarse-

grained information usually at method level which does not meet our need since we need very fine-grained analysis at code statement level. Another set of research [85, 104] utilizes dynamic analysis to understand the impact of code changes. However, dynamic analysis often falls short of code coverage and is not suitable for us either.

Static slicing [129] becomes a promising technique to grasp a comprehensive understanding of the impact for code changes. A series of research [118, 82, 144, 112, 90] has been done towards this direction. GRACE [82] proposes to perform forward slicing to capture all potentially affected codes. However, static slicing algorithm [129] is very conservative and usually generates large slices. To deal with this problem, Sridharan et.al. [118] propose a new slicing algorithm called thin slicing which only considers value-flow. P-slicing [112] and PRIOSLICE [144] augment the forward slicing with a relevance score which indicates how likely a code statement can be actually affected by the change so as to reduce the size of slices. As we can see, no existing approach can maintain the soundness for our updating purpose with respect to both control and data dependencies after reducing the size of slices.

2.3 Android Program Patching

Automatic Program Patching in the context of Android falls into two categories: Android system patching and Android app patching. A number of research works have been done [102, 51, 143, 50] to perform patching on Android system programs and kernels. PatchDroid [102] uses in-memory patching techniques to address vulnerabilities in both native and managed code. KARMA [51] is proposed as an adaptive live patching system for Android kernels. It features a multi-level adaptive patching model that can overcome the

Android fragmentation issue. Embriodery [143] only targets the binary code in Android kernels by using binary rewriting techniques. It transplants official patches of known vulnerabilities to different devices by adopting heuristic matching strategies to deal with the Android fragmentation issue. InstaGuard [50] adopts hot-patching techniques to patch the system programs in Android. It takes a different approach that enforces instantly updatable rules that contain no code to block exploits of unpatched vulnerabilities.

Android application patching techniques, on the other hand, are also proposed to mitigate security problems in Android apps. AppSealer [139], which is the most similar work with ours, performs automatic patching for preventing component hijacking attacks in Android apps. Capper [140] and Liu et.al. [92] rewrite the Android apps to keep track of private information flow and detect privacy leakage at runtime. CDRep [95] fixes cryptographic-misuses in Android with similar byte-code rewriting technique. Azim et.al. [41] detect crashes dynamically and use byte-code rewriting technique to avoid such crashes in the future.

2.4 Code Similarity Detection

As mentioned, existing technique mainly fall into three categories: static approaches, dynamic approaches and learning based approaches.

Static Approaches. Static approaches transform binary code into graphs (e.g., control flow graph) using program static analysis and then perform the comparison among binary code. Bindiff [34, 60] which is the state-of-the-art binary diffing commercial tool, performs many-to-many graph isomorphism detection on callgraph to match functions and leverages

intra-procedural CFG graph matching for basic blocks. Binslayer [47] further augments the graph matching with the Hungarian algorithm for bipartite graph matching to improve the matching results. Pewny et.al. [108] searches bugs within binary programs by collecting a list of input/output pairs to capture the semantics of a basic block and perform graph matching. The major drawback for graph matching based approaches is that the algorithms in general is very expensive. For the sake of improving runtime performance, discovRE [65] chooses to use lightweight syntax level features and applies pre-filtering before graph matching. However, the pre-filtering process may significantly affect the accuracy of the matching result [68].

Due the limitations of graph matching, techniques are proposed to use static program analysis to decompose functions into fragments. Tracelet [57] converts CFGs into a number of paths with fixed-length called *Tracelets* and then matches them by rewriting. Esh [55] decomposes the functions into segments named *strands* which represent data-flow dependencies and uses statistical reasoning to calculate similarities. GitZ [56] further improves Esh to find strands equality through re-optimization. Although no heavy graph matching is required, these techniques still bear with two major limitations. First, to abstain from massive number of fragments, they can only decompose within functions. Second, these techniques can not handle instruction and basic block reordering.

Dynamic Approaches. Based on an insight that similar code must have semantically similar behavior, dynamic analysis becomes another promising line of research. Blanket Execution [61] executes functions of the two input binaries with the same inputs and compares monitored behaviors for similarity. BinHunt [70] uses symbolic execution and theorem proving and compares intra-procedural CFGs to find the matching between basic blocks.

iBinHunt [99] extends the comparison to inter-procedural CFGs and reduces the number of candidates of basic block matching by monitoring the execution under a common input. CoP [94] also uses symbolic execution to compute the semantic similarity of blocks and leverages the longest common sub-sequence of linearly independent paths to measure the similarity. BinSim [100] which is specifically proposed to compare binaries with code obfuscation techniques, relies on system calls to perform dynamic slicing and then check the equivalence with symbolic execution. Essentially, dynamic analysis approaches could deliver accurate results when facing compiler optimizations or obfuscation. However, they by nature suffer from poor scalability and incomplete code coverage.

Learning based Approaches. Recently, researchers have turned to machine learning techniques to detect code similarity. Genius [68] forms attributed control flow graphs (ACFG) for each function and calculates the similarity through their graph embeddings which are generated through comparing with a set of representative ACFGs named *codebook*. Gemini [131] directly improves Genius by leveraging neural network to generate embeddings for each binary function. Then it trains a Siamese network for similarity detection. INNEREYE [150] regards instructions as words and basic blocks as sentences and utilizes LSTM-RNN to automatically encode the information of basic blocks and further uses a Siamese network to detect the similarity. Both Gemini and INNEREYE rely on supervised learning and requires extensive training with massive labeled training dataset. Asm2Vec [59] adopts an unsupervised learning approach by generating token and function embeddings using PV-DM model. However, it only works on function comparison but cannot perform binary diffing at block level. Also, it does not consider any program-wide CFG structural information.

2.5 Graph Embedding Learning

Graph analysis has been increasingly popular as many problems can be modeled as graphs. Generating vector representation of each node that contains important graph and node properties, a.k.a, graph embedding learning, has become widely popular. There are mainly three categories of embedding learning techniques [71]: factorization, random walk and deep learning.

Factorization. Ahmed et.al. [36] proposes a graph factorization technique to factorizes the adjacency matrix of the input graph by minimizing a loss function. GraRep [48] defines a node transition probability and proposes a k-order proximity preserved embedding method. The major drawback is scalability. HOPE [105] is similar to GraRep and preserves higher-order proximity. It fully captures transitivity and uses generalized Singular Value Decomposition to perform efficiently. TADW [134] considers feature vectors for nodes during matrix factorization. A recent work REGAL [75] is proposed to perform matrix factorization very efficiently with the consideration of node features. However, it only checks the existence of features other than considering the numeric values.

Random Walk. DeepWalk [107] is proposed to learn latent representations of nodes in a graph using local information from truncated uniform random walks. And node2vec [74] specifically designs a biased random walk procedure that efficiently explores diverse neighborhoods of a node to learn continuous feature representations of nodes.

Deep Learning. DNGR [49] proposes a novel graph representation model based on deep neural networks that can capture the graph structure information directly. SDNE [124]

designs a semi-supervised deep model that has multiple layers of non-linear functions to capture both the local and global graph structure that is highly non-linear. GCN [80] uses a localized first-order approximation of spectral graph convolutions to perform semi-supervised learning on graphs in a scalable way. Structure2Vec [53] is proposed for structured data representation via learning features spaces that embeds latent variable models.

2.6 Summary

Existing Android unpacking techniques can be put into three categories: 1). signature based approaches; 2). hooking based approaches and 3). data structure dumping approaches. None of them can be generic enough to have a whole view in multiple levels of the Android system nor can they detect unknown packers.

Security issues in Third-party libraries within Android apps have become very serious. Existing patching techniques focus on fixing specific types of security issues and do not deal with the outdatedness problem. On the other hand, TPL isolation based approaches do not solve security issues per se but merely limit the harmfulness of potential problems.

Binary diffing techniques serve as vital role for many security analyses including malware analysis and vulnerability analysis. Although existing learning-based approaches enjoy multiple advantages over the traditional graph based and dynamic analysis approaches, they still fall short of low accuracy, poor scalability, coarse granularity or require extensive labeled training data to function.

Chapter 3

Systematic Study for Android (Un)Packers

3.1 Introduction

Mobile computing has become a new frontier for the perpetual battle between cybercriminals and those who want to stop them. For years, those criminals are utilizing all kinds of malicious apps to gain undesired access to system resources [54, 72, 67], collect private user information [64, 62, 76, 147, 149], compromise data integrity [148, 93], etc. In response, various static [38, 128] and dynamic analysis techniques [62, 133] have been developed and deployed to capture their malicious activities. Such protection, however, has come under the threat of Android app packing, which becomes increasingly popular. Studies [146, 135] show that both malicious and benign apps utilize packing techniques to hide their code. The complexity in analyzing obfuscated code, as introduced by these techniques, has become a new barrier to protecting Android users. Particularly, without in-

depth understanding of these Android packers, malicious, vulnerable and plagiarized apps could easily circumvent the vetting process put in place by app markets and spread across Android devices through these markets.

Understanding packers. Despite the importance of this emerging trend (app packing), no comprehensive study, however, has ever been conducted to help the community understand the status quo of Android packing and unpacking techniques, which is crucial to building practical defense and mitigating the security risks brought in by these techniques. In this paper, we report our study on the problem, the first of this kind up to our knowledge. The study investigates a broad spectrum of Android packers and characterizes the apps utilizing them in terms of their security implications. More specifically, we seek answers to a set of security-critical questions, which *has never been addressed by the prior research*, as follows.

First of all, we want to find out how today's Android packers are being used, particularly by cybercriminals. *Are they (including commercial packing services) being abused by malware authors? How widely are the packers utilized by Android malware? What are the distributions of different commercial and custom packers across Android apps? How do the distributions change over time?*

Then, we look into technical details. *How do Android packers work? Is it very different from traditional packing? What are the security impacts when applying the packers to apps? Is it easy for malicious developers to exploit commercial services to pack their malware or plagiarized apps?*

Moving forward, we study the direction of technique development and its security implications. *Have Android packers been evolving? What are the future trends?*

Finally, we check the state-of-the-art of Android unpacking techniques. Particularly, *How do today's Android unpackers perform? Are they still effective in the presence of the most advanced packers?*

Answers to these questions can only be found through an in-depth analysis of packing and unpacking operations on Android code, to reliably identify related behaviors including those never seen before. This cannot be done by any existing Android unpackers [146, 135, 119], which can only handle *known* packing operations and have no view for behaviors at native level. Although the tools built for unpacking PC programs (e.g., Renovo [79]) could help find some new packers, they are just designed for binary code and cannot handle Java code. So far, none of the existing techniques are capable of performing the cross Java and native code analysis required for an in-depth understanding of complicated Android packing behaviors.

Our study and findings. To find answers to these security-critical questions and better understand the security implications of Android packing techniques, we developed an Android packing analysis framework called DroidUnpack based on a whole-system emulation. To reliably capture and analyze unpacking behaviors on Android, DroidUnpack has been designed to monitor at the lowest level and reconstruct Java-level execution. In this way, it can catch the intrinsic "write-and-then-execute" unpacking behaviors at either native level or Java level or both.

With the help of this analysis framework, we conducted a comprehensive study over 6 major commercial packers, 3 state-of-the-art unpackers and 93,910 Android malware samples in the wild.

3.2 DroidUnpack System

3.2.1 Key Idea

To address the unique challenges in detecting and analyzing unpacking behaviors in Android, we need to:

(1) Monitor app execution at the lowest level, so we do not miss any behaviors related to unpacking;

(2) Reconstruct Java level execution, for accurate detection and better understanding of unpacking behaviors.

To capture the intrinsic characteristics (i.e, Write-and-then-Execute) of unpacking, we will monitor the app execution at the native code level to label dirty memory regions, as well as code execution happens at both native and Java levels. In this way, we are able to detect and analyze unpacking behaviors happening at either level or in a combination of both.

To do so, we take a whole-system emulation based approach. More specifically, we run the android system and the app of interest within an emulator so that we can easily monitor all memory writes initiated by the app. Then by reconstructing the Java execution context from native execution, we are able to reliably detect the execution of unpacked code, no matter if the unpacked code is interpreted, pre-compiled, or just native code.

3.2.2 DroidUnpack Overview

To realize this key idea, we choose to build DroidUnpack on top of DroidScope [133]. DroidScope is a QEMU and VMI-based dynamic instrumentation framework

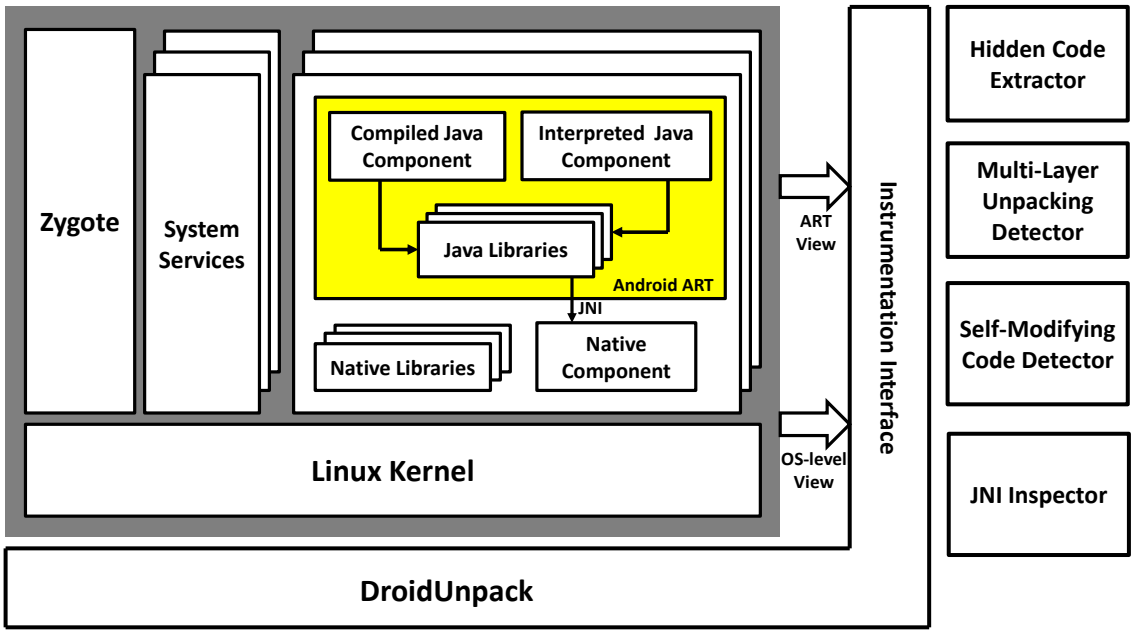


Figure 3.1: Overview of DroidUnpack.

that enables instruction tracing on both Linux and DVM sides. However, it does not support the recent Android Runtime (ART), and thus cannot recover the high-level code semantics in ART. To address this limitation, we manage to reconstruct the ART view of running Android apps. Figure 3.1 illustrates the overview of DroidUnpack.

The entire Android system, including the packed Android apps, runs on top of an emulator, and the analysis and unpacking are conducted from outside of the emulator. We introspect the guest Android system, so that both the OS-level and ART-level semantics can be reconstructed using trustworthy points-to relations among internal data structures.

Interfacing with the core DroidUnpack platform, we have developed several analysis tools to investigate packed Android programs. 1) The *Hidden Code Extractor* precisely identifies and dumps memory regions that contain hidden DEX, OAT methods. 2) The *Multi-layer Unpacking Detector* discovers iterative unpacking operations that intermittently

occur in multiple layers. 3) The *Self-Modifying Code Detector* detects an even stealthier unpacking behavior that intentionally wipes out previous executable code. 4) The *JNI Inspector* aims to search for sensitive API calls made through JNI interface.

3.2.3 Reconstructing Semantic View

Semantic view consists of two views at different levels, OS-level view and ART-level view. We rely on DroidScope to recover the OS-level view which provides two types of information: 1) the native process names and 2) the meta-data of memory-mapped modules for each process (i.e., base address, size, name, corresponding inodes and function offsets). Hence, we can accurately pinpoint a native function in memory via matching its address with $(module\_base\_address + offset)$. We modify DroidScope to support the reconstruction of ART-level semantics. In particular, we have managed to recover the application names, compiled and interpreted Java methods.

Application name. The application name plays an important role in Android app unpacking because it indicates the context of decrypted hidden code. Unlike native processes, the name of an Android application is not resolved when a `fork` takes place. Instead, its name is later appointed through calling the native function `set_process_name`. Hence, we hook this function in the native library `libcutils.so` in order to correlate application name to each individual app process.

Compiled Java method. We further recover the corresponding Java method names, offsets and sizes of native functions that have been pre-compiled from DEX code. Such information can assist accurate collection and semantic-level understanding of the code.

To collect such meta-data for compiled methods, we need to search for the associated DEX and OAT files. To do so, we first hook the function `ArtMethod::Invoke()` in `libart.so`, which accesses `ArtMethod` code for invocation. Next, at the hooking point, we retrieve the `ArtMethod` data structure from memory, which contains a reference `HeapReference<Class>` `declaring_class_` that eventually points to its host class. Using the reference, we locate the class structure that holds the resolved DEX file cache `DexCache`. Then, we can reverse engineer this DEX cache to obtain the pointer to `DexFile` data structure.

Once a `DexFile` has been discovered, DroidUnpack can further identify the code module that hosts this DEX file. This code module is in fact the OAT file that contains each original DEX file as well as its compiled `OatClass`es. By walking through each `OatClass`, we can then retrieve its meta-data, including the name and offset of every `OatMethod`. In this way, we reconstruct the mapping between name and address (i.e., $(module\_base\_address + offset)$) for each compiled method. Besides, we also iterate over every `OatMethodHeader` to find the code size of corresponding `OatMethod`, so that at runtime, we can precisely dump each unpacked code at method level.

Interpreted Java method. Although Android ART runtime provides the capability of compiling all Java methods beforehand, bytecode interpretation still remains available. Therefore, we also need to handle interpreted methods and retrieve their semantic information. Similarly, we hook the `DoCall()` function in `libart.so`, which starts the interpretation of `ArtMethod`s in Java. Again, we can trace back to the corresponding `DexFile` from each `ArtMethod`. In addition, we also obtain the `dex_method_index_` of `ArtMethod`, with which we can identify the exact `DexMethod` in the `DexFile` and therefore extract its name, offset, size

Algorithm 1 Locating Executable in Memory

```
1: procedure LocateCodeInMem(pc)
2:     mod ← GetCurrentModule(pc)
3:     if mod == "libart.so" then
4:         func ← GetCurrentFunction(pc)
5:         if func == "DoCall(ArtMethod*)" then
6:             md ← GetDexMeth(ArtMethod*)
7:         else if func == "ArtMethod::Invoke()" then
8:             md ← GetNativeMeth(ArtMethod*)
9:         end if
10:        mem_method ← GetAddressRange(md)
11:        return mem_method
12:    end if
13:    return ∅
14: end procedure
```

and bytecode instructions. In this way, we are able to capture the interpreted Java code that is unpacked during execution.

3.2.4 Code Behavior Analysis

Powered by the unique capability of DroidUnpack, we have enabled four code analyzers to understand Android packer behavior.

Hidden OAT/DEX code extraction. Malicious code is packed to avoid detection and analysis. With the reconstructed OS view and ART-level view, we can now extract packed executable code at runtime. We first follow Algorithm 1 to locate Java methods in memory. To be more specific, we examine the program counter *pc* to check whether the current running function *func* is either `ArtMethod::Invoke()` or `DoCall()`. If so, we further fetch the memory regions, mem_{method}, containing the compiled or interpreted Java method that is about to execute. In the meantime, we intercept every memory write operation to obtain the addresses of modified memory regions. As illustrated in Algorithm 2, all the dirty

memory regions are stored in $MemUD_{dirty}$, which is being continuously updated every time a memory write occurs (Ln.1). Then, Algorithm 2 detects unpacking activities by identifying mem_{method} for each basic block (Ln.12), and checking if the identified method falls into the dirty memory region (Ln.13). If that is true, unpacked code is discovered and we dump the method code and meta-data. After that, we also remove the mem_{method} from $MemUD_{dirty}$ (Ln.15), so that next time when the same method code is invoked, DroidUnpack will not count it as unpacked new code.

Note that this algorithm skips behaviors performed by Webview if JavaScript is enabled (Ln.7 to Ln.10). This filter is implemented to avoid potential false positive from JavaScript Just-in-time compilation (JIT) technique since its behavior can be mistakenly considered as packing.

Self-modifying code detection. Self-modifying code can be considered as a specific kind of unpacking. In addition to introducing decrypted new code, it also modifies the executable that has been previously launched. This practice, prevalently adopted by traditional PC malware, intends to cover the trace of historical execution or to change control flow and therefore needs special attentions.

To detect this, DroidUnpack searches particularly for the operation sequence $execute \rightsquigarrow (write \rightsquigarrow execute)$ conducted on the same memory region. Such a sequence indicates that a previously executed code region has been replaced by newly unpacked code. Algorithm 2 depicts the detection of self-modifying code as well. In addition to the aforementioned unpacking detection, this algorithm collects every executed basic-block region mem_{code} (Ln.11). The aggregation of all these code blocks, Mem_{code}, thus represents

Algorithm 2 Analysis Using DroidUnpack

```
1: MemUD_dirty ← {Overwritten memory regions updated by memory write monitor.}
2: MemUL_dirty ← {Overwritten memory regions updated by memory write monitor.}
3: layer ← 0
4: Mem_code ← ∅
5: for basic_block ∈ App execution trace do
6:     pc ← GetBeginAddress(basic_block)
7:     mod ← GetCurrentModule(pc)
8:     if JavaScript enabled and mod = "libwebview" then
9:         Continue
10:    end if
11:    mem_code ← GetAddressRange(basic_block)
12:    mem_method ← LocateCodeInMem(pc)
13:    if mem_method ∩ MemUD_dirty ≠ ∅ then
14:        DumpMethod(mem_method)
15:        MemUD_dirty ← MemUD_dirty − mem_method
16:        if mem_method ∩ Mem_code ≠ ∅ then
17:            Self-modifying code is detected.
18:        end if
19:    end if
20:    Mem_code ← Mem_code ∪ mem_code
21:    if mem_method ∩ MemUL_dirty ≠ ∅ then
22:        layer ← layer + 1
23:        MemUL_dirty ← ∅
24:    end if
25: end for
    output layer as count of unpacking layers
```

previously executed code (Ln.20). Hence, if mem_{method} is detected to be a newly unpacked method, the overlap between mem_{method} and Mem_{code} (Ln.16) demonstrates the presence of self-modification.

Multi-layer unpacking detection. Unpacking is not necessarily a one-time operation. If DroidUnpack realizes that multiple code sections have been unpacked gradually over time, it can reveal the existence of multi-layer unpacking. Concretely speaking, DroidUnpack considers all the continuously decrypted but not yet executed code belongs to the same unpacking layer, and the execution of previously unpacked code indicates the end of a layer.

Algorithm 2 shows the detection details. First, we collect another copy of dirty memory $MemUL_{dirty}$, specifically for computing unpacking layer, again via observing memory writes (Ln.2). Then, we examine the overlap between the identified Java methods mem_{method} and $MemUL_{dirty}$ (Ln.21). A non-empty intersection, indicating an execution of dirty code region is about to happen, triggers the increment of *layer* count (Ln.22) and eventually the accumulated count is provided as output. Once a new layer is discovered, we also clear the dirty memory $MemUL_{dirty}$ (Ln.23). This is to ensure that executing any unpacked code from the last layer does not cause DroidUnpack to increase the layer count.

Java native interface inspection. To avoid static inspection, sensitive APIs can be triggered through Java Native Interface (JNI) calls. Hidden bytecode or native code may also follow the same practice. Therefore, even if decrypted code has been captured, the static analysis of dumped code still may not successfully reveal the complete behavior of a packed app.

To make things even more complicated, packed apps can make recursive JNI calls. That is, a Java function $Func1$ can be invoked from a native function $Func2$ which is called through JNI from another Java function $Func3$. To handle cases like this, boundaries of each JNI call need to be captured.

Through the monitoring of context switching between Java and native modules, DroidUnpack can reliably detect the entrance and exit of each JNI calls and infer the boundaries. It further inspects all calls made at both Java and native side. In particular, DroidUnpack focuses on the detection of sensitive Android API calls invoked through JNI from native components. To identify sensitive API calls, we rely on PScout [39].

3.2.5 Discussion

Data Compression and Encoding Techniques such as data compression/encoding are not considered as packing techniques by DroidUnpack because they only introduce memory writes but do not execute at the same memory region. As a result, data compression and encoding will have no impact on our system.

Supporting Android versions. Being built upon DroidScope [133], DroidUnpack deliberately chooses to support Android 4.2 (DVM only) and 5.0 (DVM was replaced by ART) to cover the two runtime environments in Android. Supporting more Android versions will require relatively small efforts, such as recompiling the kernel and updating offsets for relevant data structures. Moreover, since variant versions of Android (e.g., Android Wear, Android Auto) share the same fundamental runtime environment, DroidUnpack should be able to support them with some fairly simple twists.

Emulation Detection. DroidUnpack is an emulation-based approach which means it cannot handle apps with emulation detection. To be more specific, our system cannot perform any automatic behavioral analysis if the apps hide all behaviors when they detect the existence of emulator. To deal with this limitation, DroidUnpack monitors four common anti-emulation techniques reported by SophosLabs [117] including examining services information, build information, system properties and the presence of emulator related files such as "/sys/qemu_trace". If any of the techniques is used by the testing app, DroidUnpack will raise alert which allows us to perform further manual investigation.

3.3 Study Methodology

To answer the four sets of research questions brought up in Section 3.1, our study of Android packer/unpacker follows a well-defined study methodology which consists of a broad range of automatic analysis using the capability of DroidUnpack as well as some manual investigations. This section elaborates on the methodology that we have systematically identified and itemized to facilitate the answers to each and every question.

3.3.1 Dataset and Setup

In order to accomplish the aforementioned tasks, we have gathered five datasets including:

- **Dataset 1**: We hand-pick seven popular and representative commercial packers including `Ali` [8], `apkprotect` [4], `baidu` [9][1], `Bangcle` [10], `ijiami` [12], `Qihoo` [14] and `Tencent` [15].

- **Dataset 2**: To study those commercial packers, we implement five representative apps, consider them as ground truth and perform diff analysis with their packed counterparts. To make sure we can seize modifications done by packers to majority of Android apps, the apps are designed to be concise yet still cover all four Android components - Activity, Service, Content Provider and Broadcast Receiver, also with two widely used features - dynamic class loading and JNI function calling. We then leverage packers in Dataset 1 to generate packed apps.

[1] `baidu` packer requires Chinese ID so we exclude it in the detailed analysis

- **Dataset 3**: For the sake of studying packing techniques among wild malware, we manage to collect 93,910 Android malware from VirusTotal [17] which are labeled as malicious by at least 50% of all detectors with a wide time span from 2010 to 2015.

- **Dataset 4**: Five recent malicious apps including Android.Malware.at_plapk.a, Android.Troj.at_fonefee.b, candy_corn, braintest and ghostpush are collected from a public malware repository in github [13] to study malware detection of commercial packers. And for plagiarized apps, we manually insert empty Android activities into three most popular benchmark apps - Vellamo, Quadrant and AnTuTu and create three plagiarized apps.

- **Dataset 5**: Lastly, we collect three state-of-the-art Android unpackers that are published in mainstream academic and industry security conferences [135, 146, 119].

3.3.2 Methodology

For each set of research questions, we elaborate our methodology by listing four most important aspects: 1). dataset, 2). challenges and solutions, 3). detailed analysis and 4). limitations. Dataset section is to describe the data samples used for answering the specific set of questions. Challenges and solutions section is to list all the technical challenges to be addressed during the study as well as our proposed solutions. Detailed analysis section describes the proposed analysis to be performed in order to explore the answer. Limitations section is elaborated to discuss the possible limitations of our analysis.

Question set 1: Are Android packers (including commercial packing services) being abused by malware authors? How widely are the packers utilized by Android malware? What are

the distributions of different commercial and custom packers across Android apps? How do the distributions change over time?

The first set of research questions is to understand the high-level landscape of current Android packers among malware, including the popularity of Android packers, distributions of each individual type of packers and how the distributions change over the years.

Dataset. In order to understand the high level landscape of Android packers, we utilize Dataset 3, the malware sample set which includes 93,910 samples in the wild with a wide time span from 2010 to 2015.

Challenges and solutions. There are two major challenges for conducting this study. First, understanding the existence of Android packers within malware samples is needed. Second, we have to further differentiate and recognize different types of packers. For the first challenge, we leverage the multi-layer unpacking detection capability in DroidUnpack to understand the existence of packing. As long as there exists a single layer of unpacking during the execution of a malware sample, we can then confirm the existence of packing within that sample. For the second challenge, as stated in [146, 135], commercial packers have strong and stable signatures across different versions. In our study, we rely on those signatures including activity names and native library names to identify the existence of different commercial packers. We collect signatures from packers in Dataset 1 and consider other packers as custom ones. Thanks to DroidUnpack, unlike previous works [146, 135], we are able to detect all the packers, commercial or custom, based on only intrinsic packing behaviors.

Analysis. To answer the first set of research questions, we first execute all malware samples in the dataset using DroidUnpack, during which we detect and record the existence and usage of different packers. We then calculate the ratio of packed malware among all the malware samples. Furthermore, we count the usage of each known packer and consider others as custom. Lastly, we extract the creation time for each sample and examine how the yearly distributions of different packers change from 2010 to 2015.

Limitations. Our analysis has several limitations. First, since we only collect signatures for the six packers, which are by no means complete, the ratio for the custom packers may be overestimated. Second, theoretically, the custom packers can impersonate the commercial packers by using the same signatures. However, we argue that the six packers are popular and representative. And despite the fact that the custom packers can impersonate the commercial packers, they probably do not have enough incentive to do so.

Question set 2: How do Android packers work? Is it very different from traditional packing? What are the security impacts when applying Android packers to apps? Is it easy for malicious developers to exploit commercial packers and pack their malware or plagiarized apps?

The second set of research questions is about detailed behaviors and impacts of Android packers.

Dataset. To understand the detailed behaviors and impacts of Android packers, we need to have ground truth first. To this end, we make use of Dataset 2 to conduct our study. We further leverage Dataset 4 to study the malware and plagiarism defense of commercial packers.

Challenges and solutions. Two major challenges need to be resolved here. First, we need to separate the behaviors of packer's code from the original code. The second challenge is to fully understand the detailed behaviors of Android packers at different levels including Java level, native level and their interactions via JNI. For the first challenge, we run our benign apps along with their packed counterparts under DroidUnpack and record all the behaviors. Then we perform diff analysis to reveal only the behaviors of packer's code. The second challenge requires us to understand the behaviors at different levels. For Java level behaviors, we rely on hidden code extractor in DroidUnpack to extract packed DEX code and further perform static analysis using other tools such as FlowDroid [38]. For native level behaviors, we are able to retrieve OS-level view and leverage self-modifying code detector and multi-layer unpacking detector from DroidUnpack to observe the unpacking behaviors. Moreover, we intercept important function calls to trace file operations, memory mapping and more to uncover how code is unpacked and loaded into the memory. For JNI interactions, JNI inspector in DroidUnpack is utilized to monitor everything that happens through JNI, especially sensitive API calls.

Analysis. In order to grasp how Android packers work, we first execute and record all the behaviors of packed apps and compare with ground truth. Then, manual investigation is performed on top of the behaviors to further understand the semantics and underlying rationale behind those behaviors so that we can not only know what happens but also why it happens. For the sake of understanding security impacts of commercial packers, we first extract the hidden code using DroidUnpack and examine the packer added code via static analysis tools and manual investigation. Lastly, we act like malicious developers to submit

malware samples and plagiarized apps to commercial packing services and check whether the submissions can be detected and prevented. To further measure the impact of packing in terms of malware detection, we submit the packed malware samples to VirusTotal [17].

Limitations. Since our analysis involves human effort to investigate the behaviors, there may be some behaviors that fail to catch our attention, therefore are missed by our study. Also, we fail to find any service that could allow us to measure the impact of packing in terms of plagiarism detection.

Question set 3: Have Android packers been evolving and how? And what are the future trends of this evolution?

The third set is a two-part question. It is related to the evolution of Android packers for learning the current status as well as forecasting the future trend.

Dataset. Evolution can only be observed via analyzing large amount of data. Thus, we use all samples including Dataset 2 and 3 for this purpose.

Challenges and solutions. One challenge here is how to define evolution. We define it as the change of complexity during unpacking process and characterize this complexity in two aspects: the number of unpacking layers and some unique behaviors that are designed to defeat existing unpackers. Inevitably, packed apps have to perform an unpacking process before original code can be executed. This unpacking process is not necessarily a one-time effort, in stead, it may contain multiple layers of packing and unpacking. Subsequently,the number of unpacking layers can be a quite representative attribute to measure the complexity of packers. Furthermore, we also propose to use new behaviors that are only discovered in recent years as a sign of evolution as well.

Analysis. To capture the Android packer evolution, we first consider the number of unpacking layers by executing all the packed malware samples and utilize the multi-layer unpacking detector in DroidUnpack to collect the layers distribution information over different years. We hope to see a clear trend of increasing layers of packing. Then, we scrutinize some novel behaviors captured by DroidUnpack that are clearly targeting unpackers and only appear in the recent years and then use those to demonstrate the evolution.

Limitations. Our current measurement of complexity is by no means comprehensive and complete, as compared to the one used for measuring the traditional PC packers [123]. We leave a more comprehensive study of complexity and evolution as future work.

Question set 4: How do today's Android unpackers perform? Are they still effective in the presence of the most advanced packers?

The last set of questions is about current Android unpackers. Due to the aforementioned complexity of Android packers, we would like to see if state-of-the-art Android unpackers can handle all the cases correctly from a design point of view.

Dataset. We utilize Dataset 5, a group of state-of-the-art Android unpackers to understand the internals of unpackers and their fundamental design limitations. To test the effectiveness, we propose to use the samples in Dataset 2 and some malware with advanced behaviors from Dataset 3 to evaluate those unpackers.

Challenges and solutions. The major challenge is to understand the designs and fundamental limitations of current Android unpackers. The solution for this challenge is to study through the literatures and the source code in order to fully understand those unpackers.

Analysis. By reviewing the literatures and source code, we perform manual analysis on the fundamental designs and limitations of each unpacker. To better understand the whole picture of current Android unpackers, we would like to conduct experiments and further compare them with DroidUnpack.

Limitations. Although the three Android unpackers are state-of-the-art tools, there may exist other tools that embrace unique designs and share different insights. We will continue this investigation in our future research.

3.4 Our Findings

In this section, we present our answers to the four sets of questions raised earlier.

3.4.1 Question Set 1: High-level Landscape

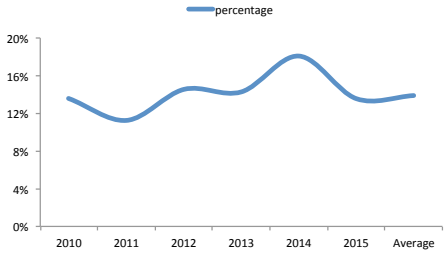


Figure 3.2: Yearly distribution.

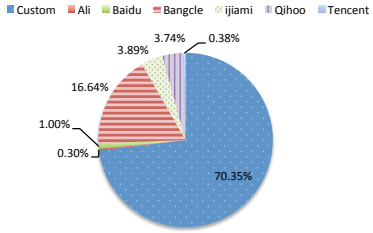


Figure 3.3: Packer distribution.

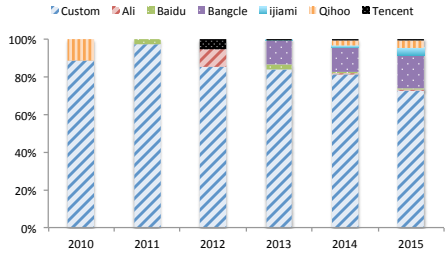


Figure 3.4: Trend of packer distribution.

As discussed in the previous reports [11, 7], researchers have found that malware samples have been leveraging packing techniques to evade detections and infiltrate into Android ecosystem. Therefore, understanding the high level landscape of packing techniques among Android malware samples has become the very first thing for us to study.

Question 1.1: Are Android packers (including commercial packing services) being abused by malware authors? How widely are the packers utilized by Android malware?

Answer: Yes, Android packers are being abused by malware author and packing techniques are quite prevalent among malware. We present Finding 1 to answer the question.

Finding 1. Malicious developers extensively leverage packing techniques to hide malice. By depicting the yearly distribution for packed apps, Figure 3.2 shows the fact that

packing techniques have embraced similar popularity among malware from 2010 to 2015 with an average of 13.89%. Interestingly, this observation contradicts AppSpear [135] where the authors claim that the ratio of packed apps is increasing. The reason to this discrepancy is that AppSpear only detects packers by signatures thus misses custom packers while we are able to extract a more complete picture of packing techniques.

Question 1.2: What are the distributions of different commercial and custom packers across Android apps?

Answer: The following finding 2 answers this question by showing the distributions of different packers.

Finding 2. Custom packed malware samples take up the largest portion of all packed malware. The distribution of packed app over different packers is presented in Figure 3.3. For all the 93,910 malware we collect, 13,052 (13.89%) of them are packed. We find custom packing takes up the biggest portion (70.35%) of the packed malware and followed by Bangcle which is utilized by 16.64% of the packed malicious apps. This finding further indicates the necessity of DroidUnpack as no existing tool can analyze custom Android packers.

Question 1.3: How do the distributions change over time?

Answer: By depicting the trend of packer distribution from 2010 to 2015, Finding 3 gives us the answer to the above question.

Finding 3. Android commercial packers are increasingly abused by malware. Apart from what has been stated above, Figure 3.4 takes one step further to illustrate the trend of packer distribution among different years. Clearly, commercial packers are increasingly

leveraged by malware as the ratio of custom packers has decreased gradually from 88% in 2010 to 69.3% in 2015. This finding is then on par with what AppSpear [135] claims.

Summary for finding 1-3. Two observations can be made from the above study. First, the existence of packed malware is a real threat with very long history tracing back to early stage of Android. This indicates that the study of Android packing techniques is not only beneficial but also necessary for malware analysis. Second, while custom packers still dominate, commercial packers are gaining popularity steadily over time. Despite the effort of enforcing different kinds of detection techniques, commercial packers still have a long way to go for filtering out malware from being packed.

3.4.2 Question set 2: Detailed Analysis on Android Packers

From the previous results, we know Android packers including commercial and custom ones have been widely abused by malware. It is important to understand the behaviors of these packers, especially the unique behaviors that do not appear in the traditional PC packers. To this end, we perform detailed analyses on both commercial and custom packers. Our study shows Android packers have embraced some unique packing techniques that are not reported by the previous Android and traditional packer research [123, 146, 135]. More importantly, as free services, we find commercial packers are not as secure and innocent as they claim to be.

Question 2.1: How do Android packers work? Is it very different from traditional packing?

Answer: Yes, Android packers are indeed very different from traditional packers. We elaborate the differences using Finding 4.

Table 3.1: Commercial packer behavior.

	apkprotect	Ali	Bangcle	ijiami	Qihoo	Tencent
Context switching via JNI	✗	✗	✗	✓	✗	✗
Native/DEX obfuscation	✓	✓	✓	✓	✓	✓
Pre-compilation	✗	✗	✗	✗	✓	✗
Multi-layer unpacking	✗	✗	✓	✓	✓	✓
libc.so hooking	✗	✗	✓	✗	✗	✗
Self modification	✗	✗	✗	✗	✗	✗
Component hijacking vulnerability	✗	✗	✗	✗	✓	✗
Information leakage	✗	✗	✗	✗	✗	✓

Table 3.2: Multi-layer unpacking.

	# of layers
apkprotect	1
Ali	1
Bangcle	9
ijiami	4
Qihoo	4
Tencent	40

Finding 4. Commercial packers have adopted many unique yet unreported features for anti-unpacking. Following the methodology described in Section 3.3.2, we comprehensively study the behaviors of six popular commercial packers. Table 3.1 summaries unique features of those packers.

App context restoration via JNI. Application context restoration is a common practice among Android packers. To hide the original code completely, packers including `ijiami`, `Qihoo` and `Tecent` create their own wrapper applications. These wrapper applications collect environment information (e.g. CPU architecture), load necessary libraries, unpack the original code and restore the app context back to the original code. `AttachBaseContext()` is the function that packers usually override to perform these tasks since it is called by the framework even before `OnCreate()` and has the ideal timing for pre-processing. JNI, on the other hand, is extensively used by packers for various of reasons. First, JNI functions

which are declared within Java level but defined in native libraries will break the control-flow and data-flow analyses. Second, some functions with heavy computations can be written as native yet still be called from Java level to boost performance. Third, packers can also leverage JNI to hide sensitive behaviors from being detected, thwarting most of the current Android app analyses. By leveraging DroidUnpack JNI analysis capability described in Section 3.2.4, we can bridge the gap between Java and native, thus understanding exactly what has happened at native level and how Java and native codes cooperate. In our study, we utilize PScout [39] and DroidUnpack to monitor sensitive API calls within JNI and discover that only `ijiami` packer cleverly invokes its application context restoration via JNI, making it harder to be detected.

Native/DEX obfuscation. As reported by the previous work [146, 135], obfuscation techniques are widely employed by commercial packers at both DEX and native levels. DEX code level obfuscation includes a wide range of techniques, such as string obfuscation, reflection, dead code injection and more. But for native code, things are slightly different. In Android, JNI builds up a bridge between Java code and native code, allowing them to interact with each other. There are mainly two ways of performing method lookup: 1) traditionally, developers could name the JNI functions in a specific way using `Java + package name + class name + function name` format so that the function mapping is automatically handled; 2) JNI functions can also be explicitly registered via `JNI_OnLoad`. All the packers other than `apkprotect` take this approach so that they can randomize function names, making it more difficult to obtain the control flow graph. Moreover, most of the commercial packers will introduce native libraries as stated in [146] for the purpose of performing

unpacking. We discover that all of the native libraries are equipped with encryption to data and code sections within binary so as to prevent analysis. At runtime, the libraries are first loaded into memory, then unpacking code will identify the address by reading from `/proc/pid/maps` and decrypt the libraries dynamically. This kind of behavior is observed using DroidUnpack through libc function interception and memory operation analysis.

Multi-layer unpacking. Many Android unpackers [146, 135, 120] depend on an assumption that there exists a clear boundary between packer's code and original code within packed apps to function normally. However, according to our observation, this assumption no longer holds. According to our study, many commercial packers turn to multi-layer unpacking strategy, meaning other than unpacking the original code at once and loading into memory, they unpack the original code layer by layer during execution. This technique will obviously render the current memory dump based unpackers useless since the dumped memory will contain mostly the unreadable packed code other than the original code. Table 3.2 shows that `Bangcle`, `ijiami`, `Qihoo` and `Tencent` adopt this unpacking technique, among which, `Tencent` is the most complex one.

Pre-compilation. Code in OAT file is allowed to be compiled into native code or remains as DEX. From the app analysis point of view, DEX code is much better than native code in terms of readability and simplicity as it preserves semantic meaning of the program. So willfully, packers want to avoid revealing DEX code as much as possible. However, as we find out in the study, completely transforming original app's DEX code into native code is such a challenging idea that all packers simply avoid. Nevertheless, we would like to see if any of the packer's code is pre-compiled into native even before the installation.

In order to detect this behavior, we first configure the `dex2oat` (the ART compiler) to be `interpret-only` so that theoretically no code should be compiled into native code at all. Then, we utilize DroidUnpack to extract hidden code from all the packed samples and check if there still exists any native code. While the answer is expected to be negative, we actually find that the sample packed by `Qihoo` packer has pre-compiled DEX code. Further looking into it gives us more details. Just like most of the packers, when packing with `Qihoo`, the packer will insert a few new components into the app. However, unlike other packers, it pre-compiles some of the packer-added DEX code into native code by invoking `dex2oat` with default configuration, ignoring the `interpret-only` flag. This technique is much less difficult than converting app's original code into native code, but can still be very useful to hinder analysis tools understanding the whole picture, especially for tools that hook Android runtime functions, e.g., DexHunter [146].

libc.so function hooking. Among many anti-debugging techniques that the packers adopt, libc modification is a very special one. We only observe this behavior with `Bangcle` packer. By analyzing memory operations, we discover that it actively modifies the libc.so module. Further inspection shows the packer tries to hook a series of important libc functions such as `read`, `write`, `open`, `mmap`, etc. This hooking behavior will disrupt many unpackers. Unpackers such as DexHunter [146] rely on libc functions like `fwrite` to dump the code from memory into files. When using these packers to unpack `Bangcle`, the app will simply crash if these libc functions are called, therefore, completely breaks the unpacking process. In order to bypass this restriction, one has to modify the unpackers and directly invoke the associated system calls instead of libc functions. This requirement certainly puts an extra

hurdle for unpacker users. This technique, on the other hand, will not affect DroidUnpack since it is based on whole-system emulation.

Question 2.2: What are the security impacts when applying Android packers to apps?

Answer: We discover severe security vulnerability and data breach[2] that some commercial packers are responsible for.

Finding 5. Android packers have led to severe security vulnerability and data breach affecting more than 1 billion users.

Commercial packers are believed by developers to be secure and only to protect intellectual property. The results of our study, however, shows that by applying some of the packers, the apps will have serious component hijacking vulnerability as well as information leakage problem.

Component hijacking vulnerability. Component hijacking vulnerability in Android is dangerous due to the fact that it allows malicious app to invoke vulnerable components and achieve a series of goals including privilege escalation and information stealing. One component within Android app can be considered as a potential target as long as its attribute "android:exported" is set to true in Manifest file. During the study, surprisingly, we notice two potential vulnerable components created by `Qihoo` packer: a content provider and a service. By examining the Manifest file, we find that the attribute "android:exported" for both components are set to be true, indicating the possibility of component hijacking vulnerabilities. Further study shows that the service is successfully launched during app execution. Since the service is fully packed, we utilize DroidUnpack to extract the hidden code and conduct a thorough investigation. Eventually, we confirm that the service is indeed

[2]This issue was identified by static analysis. We tried to contact *Tencent* to confirm but no reply so far.

vulnerable to component hijacking attacks. The service handles two different intents, one of them allows the service to download files from remote server and replace arbitrary file within the app using the app's permission. We manage to write a Proof-of-Concept code that can exploit this vulnerability by downloading a DEX file from our own server and replacing arbitrary files within the vulnerable apps. In a nutshell, using this packer to pack a perfectly secure app exposes serious arbitrary file write and even arbitrary code execution. We have reported this security issue, it was acknowledged and assigned highest priority.

Information leakage. Our study unveils another astonishing fact that one of commercial packers adds code to the original app to collect sensitive user data and send back to its own servers, thus causes an information leakage problem. As shown in Table 3.1, among the packers we study, `Tencent` packer introduces this kind of dangerous behavior. Upon packing, it will add six new permission requests to the original apps including some very sensitive ones such as `ACCESS_NETWORK_STATE` and `READ_PHONE_STATE`. Once the packed app is launched, it will collect sensitive user data such as "deviceId", "subscriberId", "MAC address", and send them back to its own server via an insecure HTTP connection. This behavior not only leaks user sensitive information to their server without any user awareness but also gets them exposed to the public as attackers can easily eavesdrop via man-in-the-middle attack. During the investigation, we rely on DroidUnpack to extract hidden code and discover the information leakage using FlowDroid [38], a state-of-the-art context-, flow-, field-, object-sensitive static analysis tool.

Impact. By simply examining the apps that are using the two problematic packers, we can draw a conclusion that these two security issues are very severe as they are affecting

Table 3.3: Security scrutiny.

	apkprotect[1]	Ali	Bangcle	ijiami	Qihoo	Tencent
Malware defense failure	5/5	0/5	2/5	2/5	0/5[2]	1/5
Plagiarism detection failure	3/3	3/3	3/3	3/3	3/3	3/3

[1] apkprotect is not on-line service and has no prevention for malware or plagiarism.
[2] Qihoo detected first attempt and blocked further malware submission.

more than 1 billion users right now. `Qihoo` packer, which introduces component hijacking vulnerability, has been leveraged by some most famous apps including Gaode Navi, Qianniuniu Finance. Gaode Navi is actively used by more than 500 million users as their daily navigation app. The vulnerability within it can easily be leveraged by attackers to obtain users' daily routing information. Qianniuniu finance, which has been downloaded for more than 3 million times, is an investment app. The vulnerability within it can severely damage users' financial security. The information leakage issue, on the other hand, is affecting even more users as it is applied by a series of popular apps including QQ, a chatting app that has more than 800 million active users.

Question 2.3: Is it easy for malicious developers to exploit commercial services to pack their malware or plagiarized apps?

Answer: Yes, Finding 6 shows that it is very easy for malicious developers to exploit commercial packers and avoid being detected.

Finding 6. Malicious developers can easily exploit commercial packers to pack malware and plagiarized apps and thus evade detections.

As we know, all of the packers except for `apkprotect` provide on-line services which aim to present packing service to protect developers' intellectual property while avoid being leveraged by malware and plagiarized apps. Consequently, they all claim to implement some

Table 3.4: Malware detection rate comparison.

Malware name	Original detection rate	Packed detection rate
Android.Malware.at_plapk.a	61.67%	26.67%
Android.Troj.at_fonefee.b	66.67%	35%
braintest	63.33%	37.29%
ghostpush*	70%	N/A
candy_corn	68.33%	38.98%

* All commercial packers can successfully detect it as malware.

kinds of security scrutiny. To this end, we conduct a study on this subject by submitting 5 confirmed recent (early 2016) malicious apps and 3 plagiarized apps to these packers and the results are presented in Table 3.3.

Malware defense. Malware detection is hard but packed malware detection is even harder [11, 5]. To protect users from being compromised, all studied commercial packing services claim to conduct advanced code analysis to rule out malware. However, our study result somehow shows otherwise. Among those five packers, `Qihoo` is namely the best when it comes to malware defense. It detected our first malware and blocked us from further submission. `Ali` also managed to detect all five malicious apps and prevented us from packing them. Together with Figure 3.4, we can clearly observe a huge improvement over malware detection for `Qihoo` and `ALi`. Other packers, however, can only detect a portion of them resulting in successful packed malware. We then submit the original malware samples as well as the packed ones to VirusTotal [17]. As illustrated in Table 3.4, the detection rates for malware have dropped significantly after packing, showing that malicious developers can easily exploit commercial packers to pack their malware and evade detection.

Plagiarism detection. Although all packers claim to help developers scan over multiple Android markets to detect plagiarism, no one actually stops developers from submitting plagiarized apps to its server. In our study, we submit three plagiarized apps to those pack-

ers and easily create packed plagiarized ones through all packers without any issue. This security loophole can be effortlessly leveraged by plagiarized app developers to pack their apps, rendering plagiarism detection more difficult.

3.4.3 Question set 3: Evolution of Android Packers

Question 3.1: Have Android packers been evolving and how? What are the future trends of this evolution?

Answer: Yes, Android packers are clearly evolving. We describe this trend with Finding 7.

Finding 7. Android packers have been evolving very fast in the last few years. Based on the systematic study of large number of packed malware samples over multiple years, we observe that Android packers are clearly evolving. We characterize this evolution in two different aspects: the number of unpacking layers and featured behaviors.

Figure 3.5: Layer distribution.

Number of unpacking layers. We have seen multi-layer unpacking in commercial packers, but we haven't seen such complicated unpacking process as shown in Figure 3.5. In year 2015, there exist about 1.3% of custom packed Android malware that unpack their hidden code with more than 1000 layers. This level of complication is never observed in commercial packers and certainly brings tremendous difficulty for unpackers to operate. In contrast, the most complicated custom packed malware we have in year 2010 has only 6 layers. The ratio of packed malware that equip with 10 or more layers unpacking has grown from 0% in 2010 to 24.73% in 2015 which is a clear indicator that Android custom packers have been evolving in a fast pace.

Behaviors. We consider two interesting behaviors as a clear sign of evolution for Android packers. First is the aforementioned libc.so hooking. As described, `Bangcle` packer modifies libc.so module so as to hook functions and prevent unpackers such as DexHunter [146]. By analyzing the timing, we can see this behavior was not added by `Bangcle` until DexHunter has released. Clearly, `Bangcle` itself is evolving to defeat unpackers. Second, a more advanced technique has been observed by us that it modifies DEX code at runtime so that apps can change their behaviors dynamically. By closely monitoring memory writes and code executions as described in Section 3.3.2, we observe this behavior in 20 out of 93,910 wild malware samples, 6 from 2014 and 14 from 2015. Self modification is normally done via JNI since native code is more suitable than Java code for memory manipulations. The app invokes JNI function which is responsible for code modification to start this process. The function first finds the right module by scanning over `/proc/self/maps` file which stores the addresses of all modules. Then, it locates OAT file in memory via

magic number "oat\n" and parses the OAT file to acquire the correct class and method to modify. Before modification can be performed, it needs to change the memory protection by calling `mprotect` to make it writable. Finally, payload is inserted into the designated code region via `memcpy` and gets executed. This kind of technique has been useful for hiding sensitive code from static analysis and unpacking tools. For example, one sample dynamically modifies the code so that other than invoking the original function, it calls a different one. Note that this technique is designed to work on DEX code which means ART may have compatibility issue as the compiler compiles DEX code into native code Ahead-Of-Time. To verify, we test those apps again with `dex2oat` configured as "speed" mode and observe that self-modifying behavior has disappeared.

Future trends of this evolution. Android packers are evolving. We believe the future Android packing technique could push its limits further into several directions that could get unpacking increasingly problematic. First, more interactions between DEX code and native code will appear in packing techniques. Native code is favorable for packers as it is unobservable from Java level, and thus is more difficult to extract. Moreover, it is a known challenge to recover semantics information even if unpackers can successfully extract the code. The pre-compilation behavior we observe is only the very first step that falls into this category. Second, Android packing strategy will continuously become more sophisticated. We have seen Android packers growing from single-layer to multi-layer and will probably see packers carrying other features as what has happened in PC packer [123], such as cyclic transition, multi-frame and more. Third, Android packers may eventually turn to emulation-based packing techniques which can defeat all existing unpackers including DroidUnpack.

3.4.4 Question set 4: Android Unpackers

We study the designs, implementations and limitations of the mainstream Android unpackers and test them against the packed malware samples in the wild.

Question 4.1: How do today's Android unpackers perform? Are they still effective in the presence of the most advanced packers?

Answer: No, state-of-the-art Android unpackers are not working properly as expected. We clarify this answer by introducing Finding 8 which gives an overview of how those unpackers perform.

Finding 8. State-of-the-art Android unpackers have serious design limitations that they cannot handle advanced Android packers. Current unpackers could be roughly categorized into three types based on distinct system designs. 1) Locate DEX file by signature and perform memory dump; 2) Modify DVM to hook certain important functions to find DEX file and then dump the code; 3) Modify DVM to dump Dalvik data structures on the air and then assemble them back into a DEX file. As discussed in Section 3.3.2, we pick three state-of-the-art unpackers from each category and compare them with DroidUnpack in Table 3.5.

Design choices. Kisskiss [119] follows a very traditional unpacking process. It is compiled as a stand-alone program and pushed into Android system for attaching to and accessing memory of target application using `ptrace`. It recognizes `odex` objects based on the memory map and the magic number and finally performs the memory dump. DexHunter [146], on the other hand, is designed to be more general-purpose based on a study of protections of current packers. Relying on customization of class loading of both Dalvik and

ART runtime, it guarantees that all classes of `odex` are initially loaded, correctly located and then extracted. Certainly, this runtime customization approach is immune to anti-debugging and anti-emulation techniques. AppSpear [135] adopts techniques of bytecode extraction and DEX reassembling based on Dalvik instrumentation. Once the main activity is interpreted or a new DEX file is loaded, AppSpear extracts the inner Dalvik Data Structure (DDS) and performs a reassembling process to recover the DEX file. DroidUnpack takes a completely different approach from those unpackers by leveraging the whole-system emulation technique. It detects a packed app via monitoring program execution on overwritten code regions and relies on only intrinsic characteristics of Android runtime and enables VMI to recover hidden code.

Limitations. Unlike DroidUnpack, none of the existing Android unpackers can have a whole view in multiple levels of the system nor can they detect unknown packers in a complete fashion.

Besides this, Kisskiss faces several severe limitations. First, commercial packers usually deploy techniques, like anti-debugging or in-memory obfuscation towards `odex` objects, to defeat this unpacking process [146]. Since Kisskiss relies on the magic number to dump `odex` objects, it does not work with unknown new packers or even the upgraded version of existing packers. Moreover, as it only dumps the memory once based on signature and could be easily defeated by more advanced techniques such as multi-layer unpacking and self-modifying code.

DexHunter mainly improves the way of locating DEX file in memory by hooking class loading functions. This design choice makes it more robust than Kisskiss. However,

it is still far from being perfect. First of all, as stated in the previous section, libc.so function hooking in Bangcle packer could defeat DexHunter unless users modify it accordingly. Second, multi-layer unpacking and self-modifying code will result in incomplete and even erroneous code dump because DexHunter only dumps the memory at the class loading time when the hooking functions get triggered.

AppSpear customizes DVM to collect the DSS data structure so that the code it dumped must be unpacked. However, it still exposes a few important limitations. First, it only works in Dalvik but not in the latest ART. In ART, code can be compiled into native during installation and will not even appear in any Dalvik data structures. Second, finding correct timing to extract DSS can be a very challenging task. By default, it only considers the main activity as unpacking point [135] which may lead to incomplete code coverage.

Despite the fact that DroidUnpack can overcome limitations described above, it does have a few drawbacks. As shared by all dynamic analysis techniques, DroidUnpack certainly suffers from limited cover coverage as it can only dump the code that executes. And since it is built on top of whole-system emulation, packers that enforce anti-emulation techniques will inevitably break the analysis.

Experiments. We conduct the experiments upon Android 4.3 and 4.4 emulators for two popular open sourced unpackers. DexHunter and Kisskiss with two datasets. 1) Dataset 2 in Section 3.3.1; 2) self-modifying malware samples collected in the above study. As shown in Table 3.5, at the time our experiment was carried out, Kisskiss failed to dump memory from all six packers. This is probably because the signatures that Kisskiss relies on have been changed. The experiment results then show that DexHunter is rather sensitive

Table 3.5: Study of unpackers

Tool	Design	Limitations	Open source	Recover code	Self-modifying samples
DexHunter	Modifies DVM and hooks class loading functions for locating and extracting DEX file	a) rely on feature string, which could vary when a packer is upgraded; b) difficult to find the right timing, can't deal with incremental packer, which means there isn't a single moment when all codes coexist in memory together	Yes	Success: Tencent. Failure: Ali, Bangcle, ijami, Qihoo. Not support: apkprotect	No
AppSpear	When MainActivity is launched or a new DEX file is loaded, AppSpear extracts inner DDS and reassembles the DEX file.	a) lack of support for ART; b) hard to find correct timing for extraction	No	N/A	N/A
Kisskiss	Uses ptrace to attach to the memory of target applications, and identifies and dump odex objects based on memory map and magic number.	a) can't handle apps with anti-debug or in memory obfuscation techniques; b) requires understanding of the specific packer to get magic number thus doesn't work with unknown packer or even slightly upgraded existing packer	Yes	Success: none. Failure: all samples. Can find odex file in memory map, but failed in locating the correct address. So, pread syscall failed.	No
DroidUnpack	Monitor program execution and memory operations based on whole-system emulation	a) cannot handle packed samples with anti-emulation	Yes	Success: Ali, apkprotect, Bangcle, ijiami, Tencent, Qihoo	Yes

in the arms race with packers. The prototype relies on a fingerprint (feature string) of each known packer, which we found only works for `Tencent` packer now. Note that the result for `Tencent` is still incomplete as it adopts multi-layer unpacking.

Chapter 4

Automatic Generation of Non-intrusive Updates for Third-Party Libraries

4.1 Introduction

Third-party libraries (TPL) have been used extensively in Android to provide rich complementary functionalities for Android apps and ease the app development. This trend becomes even more obvious recently as Android apps are getting increasingly complicated. Prior research has shown that every app contains 8.6 distinct TPLs on average [145], and 42.9% of Android apps even have more code in TPLs than in their real logic [87].

Despite the benefits, TPLs also brings serious security problems for Android app. It has been revealed [43] that 70.40% of Android apps include at least one outdated TPL

and 77% of app developers update at most a strict subset of their included libraries, leaving many known security vulnerabilities unpatched within their apps. In fact, updating TPLs in Android apps can be so time-consuming and tedious that app developers are often forced to leave TPLs outdated. First, updating libraries to the latest version is very likely to involve considerable manual efforts to solve backward incompatibility issues [58]. Second, although 97.8% of actively used library versions with a known vulnerability could be fixed via a drop-in replacement with a specific version [58], it is impractical for app developers to search for a suitable version and replace the vulnerable one for each and every library in their apps.

Existing Research. Prior efforts have been made to study and mitigate the problems with TPLs in Android apps. To understand TPLs, a variety of library detection techniques are proposed [96, 52, 43, 87, 88, 58, 125, 145] to detect TPLs in apps and study the prevalence [87, 125, 145], library evolution [88], up-to-dateness [58] and other security issues [43, 58]. Further, a series of techniques are proposed to isolate TPLs from the Android app. TPLs can be transformed into new processes [116, 142], new apps [121, 78], or new services [106]. Other works enforce in-app privilege separations [127, 114] in order to keep the apps' privileges from TPLs. However, these techniques do not fix security issues per se but merely limit the harmfulness of potential problems in TPLs from the apps.

To alleviate the issues, AppSealer [139] performs automatic patching for preventing component hijacking attacks in Android apps. Capper [141] and Liu et.al. [92] rewrite the Android apps to keep track of private information flow and detect privacy leakage at runtime. CDRep [95] fixes cryptographic-misuses in Android with similar byte-code rewriting technique. Azim et.al. [41] detect crashes dynamically and use byte-code rewriting technique

to avoid such crashes in the future. Nonetheless, these techniques only aim to fix specific types of security issues and do not deal with the outdatedness problem on TPLs. Hence, existing patching techniques on Android cannot keep TPLs updated and fix security issues in a generic fashion.

Our Approach. To solve the problem, we aim to automatically generate updates for TPLs in Android apps in a non-intrusive fashion such that it does not require any code modification on the app side and more importantly, introduce no impact to the library interactions with other components locally and remotely as we call it *non-intrusive*. The advantages of *non-intrusiveness* are two-fold: 1). it requires zero change to the code for the given Android app so that the full backward compatibility and maintainability of the apps are ensured; 2). the internal state consistency of the app is secured since the updates guarantee no impact to the program logic of the updated library.

To achieve this goal, we need to understand the impact of the code changes between the outdated libraries and the latest versions. LibBandAid utilizes forward program slicing algorithm to perform Impact Analysis [46]. Traditional slicing algorithm [129] is extremely conservative and often generates unwieldy slices [45, 113]. In our case, these slices will very likely to violate the *non-intrusiveness*. Techniques [118, 144, 112] have been proposed to prune the slices. However, they either consider only data-flow [118] or calculate relevance scores [112, 144] and remove the less relevant codes. Obviously, none of them can meet our need of soundness. As a result, we propose a novel slicing algorithm called *Value-sensitive Differential Slicing* that fully leverages the diffing information between two versions and eliminates the over-conservativeness of the traditional slicing by keeping track of value set

changes for all variables. With it, we are able to produce much smaller slices while still preserving the soundness for the purpose of updates generation.

We further implement a prototype system called LibBandAid to solve the outdatedness of TPLs. Our system first extracts the outdated libraries from a given Android app, compares each outdated library with its latest version counterpart and generates diffing information that precisely characterizes the code changes at code statement level. Then it uses our new slicing algorithm to analyze the impact of each code change and group related changes together to form a set of candidate updates based on control and data dependencies. Finally, our system carries out a selective updating process to apply only the *non-intrusive* updates to the Android app.

We then conduct a comprehensive evaluation on LibBandAid by collecting 9 popular TPLs with 173 security related commits across 83 versions and 100 real world apps. The experimental results show that LibBandAid can effectively patch the security vulnerabilities with a high success rate.

Contributions. In summary, this paper has made the following contributions:

- We propose an automatic non-intrusive patch generation technique and implement a prototype system called LibBandAid, which is the first of its kind to solve the outdatedness problem for TPLs in Android apps.
- A novel slicing algorithm called *Value-sensitive Differential Slicing* is proposed to utilize the diffing information between old and new versions of the code and reduce the over-conservativeness of the traditional forward slicing while still preserving the soundness for generating updates.

- We evaluate LibBandAid with 9 popular TPLs with 173 security related commits across 83 different versions and 100 real world apps. The experimental results show that LibBandAid can effectively update the outdated library to fix security vulnerabilities with an average success rate of 80.6% and even higher rate of 94.07% when combined with potentially patchable vulnerabilities. We demonstrate the correctness of the updated apps with automatic and manual testing.

4.2 Problem Statement

Deployment Model. Our proposed technique is anticipated to be deployed as a service for Android app developers (other than app markets or end users). Developers can feed their apps with an outdated TPL as well as the latest version of that TPL into LibBandAid. Our system will perform automatic updating by generating and applying non-intrusive updates to the TPL within the submitted Android app without any modification to the apps' code. Our approach is designed to be conservative such that it is guaranteed to maximize the updating in a non-intrusive manner. As a result, security related updates as well as other updates (e.g., new features and optimizations) can be applied to the outdated library.

It is noteworthy that the trade-off of *non-intrusiveness* is the completeness. LibBandAid will avoid applying updates that could change the interactions among the TPL and other components. As a result, our approach makes a reasonable underlying assumption so that LibBandAid is set to cover most of the security related updates.

Assumption. LibBandAid is designed to update the outdated TPLs as much as possible with a high coverage for security related updates without violating the *non-intrusiveness*.

The underlying assumption is that a security patch (e.g., insert a new condition check) is unlikely to introduce backward incompatibility or change how the TPL interacts with other components locally (e.g., with the app) and remotely (e.g., with TPL server). Hence, most of the security related issues can be fixed by our technique as they are very unlikely to be filtered out by the pre-defined rules that are designed to ensure the *non-intrusiveness*. This assumption is demonstrated by our evaluation with 9 most popular TPLs in Section 5.7.

Goals. Specifically, LibBandAid achieves the following design goals:

- **No source code required.** Our technique does not require any source code from Android app or the included TPLs. This is important because TPLs can be closed-source.
- **High coverage for security patches.** LibBandAid aims for a high coverage in updating security related issues in outdated TPLs.
- **Non-intrusiveness.** The generated updates do not change how the original app interacts with other components nor do they break the correctness of the app.

4.3 System Overview

In this section, we first present a running example and use it to explain the work-flow of LibBandAid. Note that our approach works at byte-code level, source code is presented here only for ease of understanding.

4.3.1 Running Example

The example is based on Dropbox library [19], one of the most popular third-party libraries. Assuming that a given Android app is using Dropbox library version 3.0.3 (released on May 2017). There exist 50 commits from version 3.0.3 to the latest version 3.0.6 (released on Jan 2018), including 16 code commits [1]. Listing 4.1 displays two commits. Lines with colors show the code changes: green indicates code insertions while red and yellow specify code modifications.

The first commit is a new security feature commit to add a field `accountId` in the class `DbxAuthFinish` to identify Dropbox users instead of using `userId` in older versions. The second commit is a vulnerability fix that adds a `body` field and calls `close()` function of the `body` in a callback function `onFailure()`. When Internet access is cut off, the callback function `onFailure()` will be invoked to close `body` so that potential system hang is avoided.

4.3.2 Overview of LibBandAid

Figure 4.1 delineates the overview of LibBandAid. As shown, there exist four major components in LibBandAid: preprocessing, diffing analysis, update generation and selective updating.

[1] Other non-code commits include changes in README, build file, tutorial and tests

Listing 4.1: Running example

```
1  public class DbxAuthFinish implements CallBack {
2    private String userId;
3  + private String accountId;
4  + private PipedRequestBody body;
5
6  - public DbxAuthFinish(String uid){
7  + public DbxAuthFinish(String uid,String aid,Body body) {
8        this.userId = uid;
9      + this.accountId = aid;
10     + this.body = body;
11   }
12   public DbxAuthFinish(){
13     + this.body = null;
14     + this.accountId = null;
15       this.userId = null;
16   }
17   public void onFailure (IOException ex) {
18       this.error = ex;
19     + if(body) this.body.close();
20       notifyAll();
21   }
22   public DbxAuthFinish read() {
23     + String accountId = null;
24       String userId = null;
25
26       while(getCurrentToken()) {
27           if(n.equals("uid"))
28             userId = readField();
29         + else if(n.equal("accountId"))
30           + accountId = readField();
31
32         + if(accountId == null)
33           + throw JsonReadexception;
34       }
35     - return new DbxAuthFinish(userId);
36     + return new DbxAuthFinish(userId, accountId, body);
37   }
38 + public String getAccountId(){
39   + return accountId;}
40 }
```

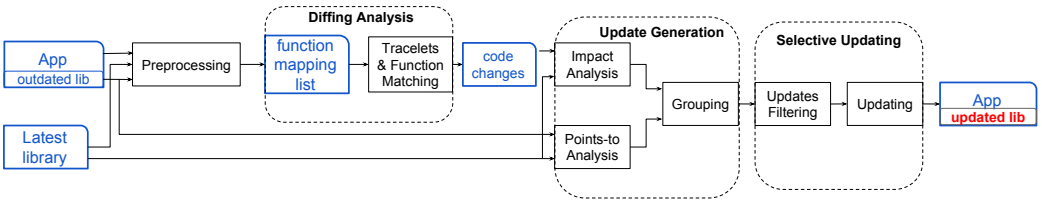


Figure 4.1: Architecture Overview.

Preprocessing. This step is to filter out the unchanged functions and generate function pairs that are modified across the two versions. Preprocessing component takes as inputs an app with outdated library and a latest version of the library, and outputs a set of function pairs. More specifically, it extracts the outdated library within the given app, analyzes all classes in the two versions of the library and performs function level byte-by-byte comparisons.

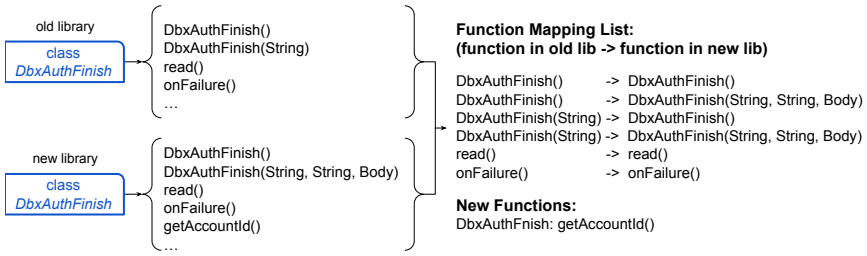


Figure 4.2: Preprocessing.

As shown in Figure 4.2, LibBandAid first pulls out all the functions in the class and performs byte-by-byte comparisons for each function in old library with the functions in the new library as long as they share the same function name. Note that we use function name other than function signature to tolerate changes of modifier, parameter or return type. For example, `DbxAuthFinish()` in the old library is compared with `DbxAuthFinish()` and `DbxAuthFinish(String, String, Body)` in the new library. When the byte-by-byte

comparison fails (two functions are not identical), we put them in the potential function mapping list and send it to diffing analysis for further analysis. This list signifies the functions in which the code changes between old and new versions reside.

Diffing Analysis. Diffing analysis in LibBandAid is to perform function level matching with a granularity of code statement so as to comprehend the exact code changes between old and new versions of a given library. To achieve this goal, we leverage the Tracelet Execution [57] idea and use 3-tracelet to perform code matching at code statement level. Given the output of preprocessing, 3-tracelets are generated to capture partial flow information by breaking down the control-flow graphs for each function pair. Then, the distance between tracelets are calculated to match code statements.

For LibBandAid, we need to perform one more step to match the functions that have more than one matched candidates. For example, in Figure 4.2, `DbxAuthFinish()` in the old library can be matched to either `DbxAuthFinish()` or `DbxAuthFinish(String, String, Body)` in the new library. To understand real code change, LibBandAid leverages the distance information to further match the functions. Particularly, LibBandAid considers it as a linear assignment problem and uses Hungarian Algorithm [83] to find the optimal matching.

Tracelet technique has demonstrated a 0.99 accuracy in comparing functions in binary code [57]. In our case, byte-code matching is easier than binary code since it is more semantic-rich. Therefore, we observe no false positive during diffing.

In our running example, `DbxAuthFinish()` and `DbxAuthFinish(String)` are matched to `DbxAuthFinish()` and `DbxAuthFinish(String, String,Body)` in the new library respec-

tively. The final output of diffing analysis is the real mapping of the functions as well as a set of code changes (pairs of code statements) that precisely characterize the changes between the old and new versions of the third-party library. For our running example, the produced code changes are the same as the colored lines in Listing 4.1.

Update Generation. Once LibBandAid identifies all the code changes between the old and new versions of the library, it starts the update generation as depicted in Figure 4.3.

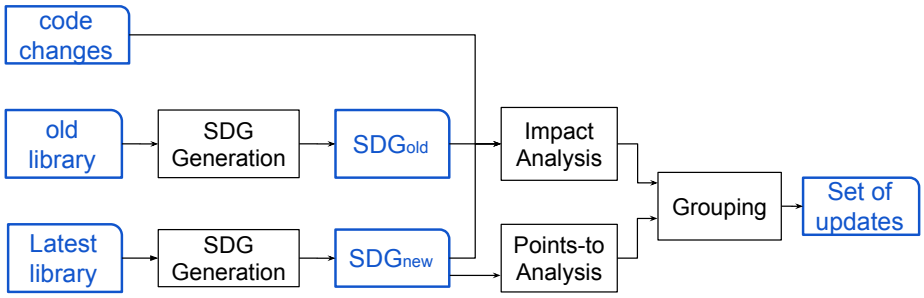


Figure 4.3: Update Generation.

The whole process takes three inputs: 1). the set of code changes generated by diffing analysis; 2). the old version of the library; and 3). the new version of the library, and generates one output (a set of updates). It first generates system dependence graphs (SDGs) for new and old library. Then it generates a patch for each code change by performing impact analysis. Finally, it performs grouping based on the alias information generated from points-to analysis to produce a set of updates.

The purpose of this indispensable step is two-fold. First, since many code changes have control and data dependencies with each other, LibBandAid should always put them together and perform updating collectively. For example, in Listing 4.1, Ln.10 and 14 assign values to a newly added class field `body` (defined at Ln.4). Ln.21 further calls a member

function `close()` of the field. These code changes should be put into one `group` since they are the definition and usages of a same variable `body`. Second, to fulfill the non-intrusiveness design goal as described in Section 4.2, LibBandAid will perform impact analysis, combine code changes with all the potentially affected codes and further associate the `group` into one update so that our system can apply them as a whole if the update is indeed *non-intrusive*. As for our running example, after this step, the code changes in Listing 4.1 will be grouped precisely into two updates, one for each commit. More details on how LibBandAid performs impact analysis and update generation will be presented in Section 4.4 and 4.5.

Selective Updating. The last component of LibBandAid is selective updating. This component takes the updates generated in the previous step, performs filtering to discard the updates that could potentially break the non-intrusiveness and eventually updates the old library to generate a new app with an updated library. The core part of this step is to systematically devise a set of pre-defined rules for filtering so that the non-intrusiveness of our generated updates can be preserved. As for the running example, two updates are generated and fed into selective updating. The one related to `accountId` can potentially be filtered out by LibBandAid since it will change an interface `DbxAuthFinish(String)` and may cause incompatibility issue. We confirm this by analyzing the given Android app. More detailed information is presented in Section 4.6.

4.4 Update Generation

In this section, we describe how LibBandAid performs update generation by presenting the three major steps: impact analysis, points-to analysis and grouping.

4.4.1 Impact Analysis

Impact Analysis is to understand the impact (affected codes) of the code changes generated from diffing analysis. Once the impact of the code changes is known, LibBandAid groups code changes into updates and performs filtering to remove the ones that violate the *non-intrusiveness*.

Program slicing technique seems to be a perfect solution. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior [130]. Hence, if we start slicing from a specific code change, it will conservatively includes all the codes that can potentially be affected by the change. However, traditional slicing is too conservative to be practical and generates gigantic slices. The larger a slice is, the more codes it contains, hence, the bigger chance it will violate the *non-intrusiveness* and get filtered out (more in Section 4.6). To solve this problem, a new slicing algorithm is desired to perform a sound impact analysis with respect to our definition of *impact* to achieve *non-intrusiveness* while greatly reducing the over-conservativeness. We discuss the slicing in detail in Section 4.5.

4.4.2 Points-to Analysis and Grouping

After the impact analysis, LibBandAid performs points-to analysis to extract alias information and further groups code changes into updates. This step is to group slices that are accessing the same global variables or have overlapping code statements. We rely on the existing points-to analysis in Soot [3] to extract alias information.

4.5 Value-Sensitive Differential Slicing

In this section, we first introduce some important definitions and then describe how our slicing algorithm works in detail.

4.5.1 Formal Definitions

We formally define the *impact* of a code change and then lay out our definitions on the relationships between program behaviors and value sets, upon which the soundness of our slicing algorithm is built.

Definition 1. We denote *impact* of a code change on a code statement as $I(d, c)$, where

- d represents a code change in the new library;
- c represents a code statement that has not changed from the old to the new version of the library;

Therefore, $I(d, c) \neq \varnothing$ means that a code change d has *impact* on code statement c. Intuitively, $I(d, c) = \varnothing$ means that a code change d has no *impact* on c. We then define a code change that has no *impact* on a code statement as:

Definition 2. $I(d, c) = \varnothing \iff B_c^d \subseteq B_c$, where

- B_c^d is a set of behaviors representing all possible program behaviors of c **with** d applied;
- B_c is a set of behaviors representing all possible program behaviors of c **without** applying d;

Here, the *impact* of a code change to a certain code statement is represented by the change of *program behaviors* for that code statement. It shows that if and only if all the possible program behaviors of a code statement c with the code change d applied are still within the original behavior set, we can say d has no impact on c.

We further make the following definition on the relationship between value-set [44] of all the variables used in one code statement and the program behaviors of that code statement:

Definition 3. $VS^d(I, c) \subseteq VS(I, c) \Rightarrow B_c^d \subseteq B_c$, where

- $VS^d(I, c)$ denotes the value set of all the variables I (global and local) and their combinations used in a code statement c **with** d applied;
- $VS(I, c)$ denotes the value set of all the variables I (global and local) and their combinations used in a code statement c **without** applying d;

Essentially, this definition shows that if the value sets of all variables and their combinations used in a code statement are unchanged or a subset of the original value sets, then the program behaviors of that code statement must stay unchanged or a subset of the original ones.

This definition gives a strong mapping from value sets of all variables in a code statement to the program behaviors of that statement. Together with Definition 2, we can draw a link between value sets of all variables in a code statement and the impact of a code change to that code statement. More specifically, our impact analysis can remove the over-conservativeness by examining the value set changes of all variables in a code statement between old and new versions of the library. If the value sets are unchanged or become a

subset of the original set for a statement before and after applying a code change, that means the code change has no impact on the statement and our algorithm can safely stop further slicing.

This may be counter-intuitive at first glance. For example, if after applying code change d, statement c has only one behavior in its behavior set while the original behavior set has 100 behaviors, d would still have **no** impact on c as long as the one behavior is within the original behavior set. However, in this case, We can actually stop slicing safely since we know the original code c can correctly handle d and its affected behavior (it is within the original behavior set and introduces no unexpected behavior).

4.5.2 Basic Scheme

The idea of our algorithm is to take into consideration the value changes of all variables between old and new versions of the code and leverage this information to reduce the over-conservativeness of the traditional slicing.

Intuitively, the basic scheme algorithm starts slicing from a code change and performs whole library-wise context- and flow-sensitive value-set analysis [44] on all variables and their combinations for each code statement that has either control or data dependency with the code change. Then it compares the value sets for the variables within these code statements in two versions of the library. If there exists no change in the value sets, meaning the code change has no impact on the current code statement, then it does not include the code statement in the slice. Note that since many values can never be determined in static analysis, we compute value formulas in a context- and flow-sensitive fashion as the value-set for non-constant variables.

Theoretically, this Impact Analysis is sound with respect to the definition of *impact*. Accordingly, it should be able to remove the over-conservativeness of traditional slicing algorithm. However, this design clearly introduces a huge performance overhead for the whole library-wise context- and flow-sensitive value-set analysis on all variables and their combinations on every control or data dependent code statement for a single code change (note that there can easily be thousands of code changes between two versions), rendering the algorithm practically infeasible.

Consequently, we present two optimizations to this basic scheme to improve the runtime performance as well as to further reduce the over-conservativeness. Again, source codes are listed just for ease of presentation while LibBandAid works on bytec-code.

4.5.3 Slice-wise Value-set Analysis

To reduce the complexity, we propose an optimization to narrow down the search space to the current slice which begins from the code change.

Listing 4.2 shows a real-world security commit from a popular library EventBus [20]. At Ln.4, a condition check `!subscriptions.isEmpty()` is added in the new version. The traditional forward slicing will start from the code change and include every single line from Ln.4 to Ln.23 and even more codes in functions like `invokeSubscriber()` since they all have dependency with the code change. However, by manual investigation, we know the code change does not introduce any new behavior to the `postToSubscription()`, hence, our slicing could stop here.

For basic scheme, we then compute value sets for all variables and their combinations in every code statement that is data-dependent on the code change. For instance, for

code at Ln.6, we calculate value sets for variables `sc` and `event` as well as their combinations (say, `sc` = 1 only if `event` == 0). This calculation can only be done in a heavyweight whole library-wise context-sensitive fashion as the value of `event` is from the caller function `postSingleEvent()`.

To accelerate the process, we can perform the value-set analysis only within the slice instead of the whole program since our analysis is to include all code statements that the starting of the slice (a code change) has affected. That is, as long as the code change (Ln.4) does not affect the value sets of `sc` or `event` or their combinations, we could stop the value-set analysis and keep our slicing from further propagating into `postToSubscription()`. This analysis can be done much faster within the current slice other than the whole library. As a result, a much smaller slice (Ln.4-8) will be produced in a very lightweight fashion.

This optimization is an approximation to the basic scheme algorithm. It sacrifices precision of the whole library-wise value-set analysis but greatly improves the performance. Consequently, it is more conservative than the basic scheme. For example, consider a case where an assignment `a = 1` is inserted in a new library. Every code that uses the variable `a` will be included under our optimization. However, a library-wise value-set analysis may tell us that `a = 1` is still within the original value-set. Therefore, we in fact do not need to include the code statements that are data-dependent on the newly inserted assignment.

Listing 4.2: Slice-wise Value-set Analysis

```
1  void postSingleEvent(Obj event) {
2      subscriptions = subscriptionsByEventType.get();
3      if (subscriptions != null
4          + && !subscriptions.isEmpty()) {
5          for (Subscription sc : subscriptions) {
6              postToSubscription(sc, event);
7          }
8          subscriptionFound = true;
9      }
10     ...
11 }
12 void postToSubscription(Subscription s, Obj event) {
13     switch (s.threadMode) {
14     case PostThread:
15         invokeSubscriber(s, event);
16         break;
17     case MainThread:
18         mainThreadPoster.enqueue(s, event);
19     ...
20     }
```

4.5.4 Intra-procedural Value-set Analysis.

As discussed, the first optimization that searches only within the slice may bring over-conservativeness. As a result, we propose a second optimization to relax the search scope of value-set analysis to the beginning of the function that contains the code change.

Consider Listing 4.3 from Dropbox [19] library. It shows another real-world security commit that fixes Android Fake ID vulnerability. Code statements at Ln.15-17 in the old version are updated to codes at Ln.19-20 in the new version and Ln.23 is updated to Ln.24. Since `return true` (Ln.17) has now become `return false` (Ln.20). Apparently, the value set of variable in the return statement has changed. According to the first optimization, our slicing algorithm will continue flowing into the call site of `hasDropboxApp()` at Ln.2, further

propagate to Ln.2-5 and eventually include almost every line of code in the example except for Ln.10-14.

In fact, a closer look will tell us that the code changes within `hasDropboxApp()` does not really expose any *impact* on its caller `onResume()`. Although the return value is modified, both the old and new versions of the function bear the same function-wise return value set: {`true`, `false`}. In order to capture this information, our algorithm needs to perform intra-procedural Value-set Analysis beyond the scope of a slice but still within `hasDropboxApp()`, which is the function that contains the code changes. As a result, our algorithm will stop slicing and the generated slice contains only Ln.19 and 20.

Listing 4.3: Intra-procedural Value-set Analysis

```
1  void onResume() {
2      if (hasDropboxApp(officialAuthIntent))
3          startActivity(officialAuthIntent);
4      else
5          startWebAuth(state);
6  }
7  boolean hasDropboxApp(Intent intent) {
8      ResolveInfo infos = queryIntent(intent);
9      if(infos == null)
10         return false;
11     else {
12         for (Signature sig : packInfo.sigs) {
13           - for(String dbSig : DROPBOX_SIGS)
14             - if (dbSig.equals(signature))
15                 - return true;
16
17             + if (!DROPBOX_SIGS.contains(sig)
18                 + return false;
19         }
20         ...
```

From the description above, we can see that this optimization sits between the basic scheme (whole library-wise context- and flow-sensitive analysis) and the first optimization

(pure slice-wise analysis). Therefore, by applying this optimization to all the variables, our slicing will be more accurate while maintaining the similar performance gain from the first optimization with negligible overhead.

4.5.5 *Value-sensitive Differential Slicing*

Now we present the details of our slicing algorithm in Algorithm 3, which is a dependence graph based slicing algorithm as [77]. It takes as inputs three elements (a code change $diff$ and SDGs for the two versions of the library SDG_n and SDG_o) and generates slice for that code change as output.

The algorithm first locates the $diff$ in two SDGs (Ln.6) and adds $stmt_n$ into a *workingSet* (Ln.7) to start the iterative process. The algorithm will continue running as long as the *workingSet* is not empty (Ln.9). For every statement in the working set, we extract its immediate successors by calling $ImmediateSuccessors()$. For every immediate successor *succ*, the algorithm checks if it is another code change. There exist two cases under this scenario. First, if *succ* is a code change which contains a new function invocation, our algorithm needs to leverage traditional slicing by calling $Forward\_Slicing()$ to keep track of the new function call (Ln.13-14) as all its codes are new codes compared to the old version. Second, if *succ* is a normal code change, we then consider it as another input to a recursive function call for $V\_Slicing()$ (Ln.15-16).

When *succ* is not a code change, we add it into the *workingSet* as well as the *slice* if it is only control-dependent on *stmt* (Ln.17-19). When *succ* is a return statement, we apply the second optimization discussed in Section 4.5.4 by performing function-wise

Algorithm 3 *Value-sensitive Differential Slicing*

1: **input1:** $diff \leftarrow \{stmt_o, stmt_n\}$
2: **input2:** $SDG_n \leftarrow$ {SDG of the new library.}
3: **input3:** $SDG_o \leftarrow$ {SDG of the old library.}
4: **procedure** $V\_Slicing(diff, SDG_n, SDG_o)$
5: $slice \leftarrow \varnothing$
6: $f_n \leftarrow Locate(stmt_n, SDG_n)$; $f_o \leftarrow Locate(stmt_o, SDG_o)$
7: $workingSet \leftarrow workingSet \cup stmt_n$
8: $slice \leftarrow slice \cup stmt_n$
9: **while** $workingSet \neq \varnothing$ **do**
10: $stmt \leftarrow workingSet.remove()$
11: $Set_{succs} \leftarrow ImmediateSuccessors(stmt, SDG_n)$
12: **for** $succ \in Set_{succs}$ **do**
13: **if** *succ* contains new invocation **then**
14: $slice \cup \leftarrow Forward\_Slicing(succ, SDG_n)$
15: **else if** *succ* is another $diff'$ **then**
16: $slice \cup \leftarrow V\_Slicing(diff', SDG_n, SDG_o)$
17: **else if** *succ* is control-dependent on *stmt* **then**
18: $slice \leftarrow slice \cup succ$
19: $workingSet \leftarrow workingSet \cup succ$
20: **else if** *succ* is return statement **then**
21: **if** $!(RetVS(f_o) \subseteq RetVS(f_n))$ **then**
22: $slice \leftarrow slice \cup succ$
23: $workingSet \leftarrow workingSet \cup succ$
24: **end if**
25: **else if** *succ* is only data-dependent on *stmt* **then**
26: $vf_n \leftarrow CalVS(succ, slice, SDG_n)$
27: $vf_o \leftarrow CalVS(succ', slice, SDG_o)$
28: **if** $!(vf_n \subseteq vf_o)$ **then**
29: $slice \leftarrow slice \cup succ$
30: $workingSet \leftarrow workingSet \cup succ$
31: **end if**
32: **end if**
33: **end for**
34: **end while**
35: Return *slice*
36: **end procedure**

value-set analysis for all return statements to improve the accuracy (Ln.20-23). When *succ* is data-dependent on *stmt*, we calculate and compare the value-sets by calling $CalVS()$ to extract value formulas at the scope discussed in the second optimization for both old and new versions and only add *succ* when *stmt* has impact on it (Ln.25-30). Eventually, our algorithm produces a slice by returning *slice* (Ln.35).

4.6 Selective Updating

The final component in LibBandAid is the selective updating. It takes the generated updates, performs filtering and applies the updates to eventually produce an updated TPL. Figure 4.1 shows the two major steps: filtering and updating.

4.6.1 Filtering

This step is to filter the updates that may affect the interactions between the library and other components in order to achieve the *non-intrusiveness* goal as explained in Section 4.2.

LibBandAid applies a set of pre-defined rules to filter out the generated updates that may violate the *non-intrusiveness*. These rules are defined to be conservative and can guarantee that all satisfying updates will not change how the library interacts with other components. To this end, we investigate into how TPLs works and propose four categories of interactions as well as the rules.

Interaction with the given app. The first category is listed in the first row in Table 4.6.1. It defines the rules for interactions with the given app. Android apps that use a TPL are by

Table 4.1: Pre-defined Rules for Filtering

Categories	**Representative Behaviors**		**Rules**
Interaction with the given app	API changes	public API signature change (return type, parameter, etc)	depend on analysis
		exception thrown change (new exception type)	depend on analysis
Interaction with server	protocol changes	incoming message change	F
		outgoing message change	F
Interaction with Android system	new Android API usages	no permission change	T
		new permission needed	F
	file manipulation	new file creation	T
		file access that modifies file pointer	F
		new file write	F
	kernel object change	thread/process creation	T
Interaction with other apps	communication to other components	new intent	F
		intent modification	F
	services	start/bind/unbind services	F

nature the most important component for the TPL to interact with. Therefore, when the library gets updated by LibBandAid, we guarantee the interactions with the app will not be affected.

Since the interactions are always through library APIs, we need to make sure the APIs that are utilized by the app will stay the same in terms of function names, return types, parameters and exceptions. To this end, LibBandAid performs program analysis to collect the library APIs used within the app and filters the updates that could change these APIs. Additionally, LibBandAid performs analysis on the library APIs to collect exception thrown information. If an update introduces a new exception, it will be discarded.

It is noteworthy that the interaction with the given app is the only category that relies on program analysis due to two reasons. First, we need to perform program analysis on the two versions of the library to understand which APIs are changed. Second, even if some APIs are indeed changed in the newer version, we may still safely update as long as the Android app does not directly call the APIs.

Interaction with server. Another important interaction for a TPL is to communicate with its server. For example, Dropbox library communicates with Dropbox server to access files. Therefore, our system needs to make sure that the protocol between server and client stays the same. To do so, LibBandAid scans over each update and checks if there exists any code within it that performs any network communication (incoming or outgoing). As long as such code exists, our system will be conservative and choose to not apply this update. For example, if one update contains API calls such as `HttpURLConnection: getResponseMessage()`, LibBandAid will filter it out.

Interaction with system. We then consider the interactions between a library and the underlying Android system.

First, our update may interact with the Android framework by calling a new Android API that does not exist in the old version. We rely on the result of PScout [39] to check if the new Android API requires any new Android permission. If it does, then LibBandAid will discard the update. Second, we check if an update performs any file manipulation in the Android system. Particularly, LibBandAid cares if the update affects the current system state, such as creating a new file or writing into a file. The tricky part is the file read. Our system only prevents the library from modifying the file pointer while reading a file (e.g., a call to `RandomAccessFile: seek()`). Third, library may create new kernel objects such as thread and process. LibBandAid allows this kind of interactions since they do not affect the execution of Android apps.

Interaction with other apps. The last category of interaction is the interaction with other apps in the Android system. Apps within an Android system could communicate with each other via Binder. LibBandAid disallows any update to change the communication either by creating a new intent or by changing any of the existing intent. Also, an update that starts, binds or unbinds services in the system is discarded.

4.6.2 Updating

After filtering out the unsatisfying updates based on our rules, LibBandAid applies the satisfying ones to the outdated library. This step is done at Jimple IR level by using byte-code rewriting capability in Soot [**?**]. After the rewriting, we convert the up-

dated Jimple IR into Dalvik byte-code, repackage the DEX file with other resource files and eventually create a new Android app (APK file) with updated library.

4.7 Evaluation

In this section, we conduct experiments to evaluate LibBandAid with respect to its effectiveness and correctness. In particular, we first study how well LibBandAid performs updating to the older versions of the libraries using a representative set of Android TPLs and real-world apps. And then we evaluate the effectiveness of our new slicing algorithm by comparing with traditional slicing algorithm in actual updating.

4.7.1 Dataset and Configuration

We collect 9 popular Android third-party libraries [43] including Butterknife [18], Dropbox [19], EventBus [20], Glide [22], Gson [23], Leakcanary [24], Okhttp [25], Picasso [26] and Retrofit [27], with a total of 173 security commits over 83 different versions to evaluate our system. Table 4.2 shows the library names, total number of security commits as well as the associated library version spans.

We first collect ground truth based on commit information provided in Github repositories to gather the vulnerability information for all the 173 security commits. Vulnerability types proposed in prior research [91] to these security related commits are presented in Table 4.3. As shown, our representative dataset covers a wide range of different types of vulnerabilities.

Then, we compile each libraries into a number of testing versions with two requirements: 1). each testing version contains at least one security commit; 2). these testing

Table 4.2: Overview of TPLs in Evaluation

Library	Security Commits	Testing Versions	Versions Span
Butterknife	6	6	7.0.1 - 8.0.1
Dropbox	11	10	3.0.0 - 3.0.6
EventBus	15	10	2.1.0 - 3.1.0
Glide	22	10	4.4.0 - 4.6.1
Gson	13	10	2.2.4 - 2.8.2
Leakcanary	42	7	1.3.1- 1.5.4
Okhttp	26	10	3.7.0 - 3.10.0
Picasso	19	10	1.5.3 - 3.0.0
Retrofit	19	10	2.0.0 - 2.4.0

versions cover all the security commits and version numbers that are listed in Table 4.2. Finally, we develop Android apps that utilize these testing versions. For each testing version other than the latest one, we feed the Android apps with these versions along with the latest version of each library into LibBandAid for evaluation. To illustrate, Butterknife library has 6 security commits from version 7.0.1 to 8.0.1. We compile 6 testing versions v1 to v6 to guarantee each one will contain at least 1 commit. Then we develop 5 Android apps a1 to a5 that use testing versions v1 to v5 and feed (a1,v6), (a2,v6),..,(a5,v6) into LibBandAid for experiments.

Furthermore, we collect 100 real-world Android apps from F-Droid [21] to demonstrate LibBandAid in practice.

4.7.2 Effectiveness of LibBandAid

As discussed, we feed each Android app with an older version library along with the latest version into LibBandAid and then manually investigate the updated libraries to see if the commits have been updated.

Table 4.3: Security Fixes Distribution

Vulnerability \ Library	Butterknife	Dropbox	EventBus	Glide	Gson	Leakcanary	Okhttp	Picasso	Retrofit
Improper Input Validation	1	3	3	6	5	2	7	6	1
Data Handling Error	4	4	5	3	3	3	7	1	6
Uncaught Exception	1	1	3	4	1	2	7	2	7
Memory Leak			1		1	32	1	3	
Info Leak						2			1
Race Condition			3						
Improper Access Control				2					
Uncontrolled Resource Consumption				1					
System Hang		1		1		1	2		
Uncheck Return Value				5					2
Illegal Reflective Access					1				
Stack Overflow					2			5	
Heap Access Error							1	1	1
Missing Initialization							1		1
Integer Overflow								1	
Fake ID		1							
New Security Feature		1							
Total	6	11	15	22	13	42	26	19	19

Security commits can be divided into three categories: 1). 'patched' means our system can successfully update the library with the commit; 2). 'fail to patch' gives the number of commits that are filtered out by the filtering process due to the violation of our pre-defined rules; 3). 'potentially patchable' shows the number of commits that change the APIs of the library. LibBandAid may still update the 'potentially patchable' ones as long as the analyzed Android apps do not directly invoke the changed APIs. Therefore, whether or not our system can update them is on a per app basis.

By Absolute Numbers. Figure 4.16 gives the results in absolute numbers for the 9 libraries. The x-axis shows each execution of LibBandAid while y-axis is the absolute number of vulnerabilities. For example, the x-axis in Figure 4.8 gives the 9 executions from (a1,v10) to (a9,v10) for Dropbox library and the y-axis shows the total number of security commits to be updated for each run. By looking at the first bar in the figure, we can see that there are total of 11 vulnerabilities between the old and new versions of the library. LibBandAid is able to fix 7 of them but fails in 2. Moreover, there are 2 'potentially patchable' security commits that change the APIs. From the 9 figures, 2 libraries (Butterknife and Picasso) are shown to have no 'fail to patch' commit (no yellow bar) for all the versions. And for the rest 7 libraries, 'fail to patch' commits only take up a very small average potion of total numbers across all executions. (a9,v10) execution in Okhttp (Figure 4.13) is the worst case in our evaluation in which it has 1 'fail to patch' commit in total of 3 commits. Further investigation shows that this is due to potential protocol change since Okhttp is an HTTP client and performs considerable amount of network communications. A more interesting observation is that the 'fail to patch' commits will disappear in many libraries

when the outdated library becomes more recent and closer to the latest version. For Gson library in Figure 4.11, starting from (a5,v10), the 'fail to patch' commit is gone.

From the experiments, LibBandAid could achieve an average success rate of 80.6% for updating security commits and even a higher rate of 94.07% when combining with the 'potentially patchable'.

By Vulnerability Categories. We then examine the categories of vulnerabilities that LibBandAid fails to update and the results are exhibited in Table 4.4 in Appendix, which shows the breakdown of vulnerabilities and the number of failures for that security commit if LibBandAid fails to update in all executions.

We find that among all kinds of security vulnerabilities, Info Leak is most likely to fail (1 failed in 3 total commits). In general, vulnerabilities that are related to IO exceptions and information processing (e.g., input validation, data handling) also bear relatively high failure rates. This result is expected since the updates to these vulnerabilities are most likely to affect the interactions between the library and the system or the server, therefore, triggering the filtering in LibBandAid.

Observations. Two observations can be made from the above experimental results. First, our assumption made in Section 4.2 that security patches are unlikely to introduce backward incompatibility or change how the TPL interacts with other components, holds in practice. Second, LibBandAid performs better in updating relatively newer version of the library. This is because the newer the library is, the less code changes it has compared to the latest version. As a result, fewer and smaller slices will be generated and they are less likely to be filtered out by our filtering process.

4.7.3 Correctness of LibBandAid

The correctness of LibBandAid is demonstrated by performing random testing as well as manual investigation for the apps that are updated by our system. To this end, we first use LibBandAid to update TPLs within the 100 real-world apps from F-Droid. Then, we collect apps with updated TPLs for testing.

For random testing, we run Monkey, which is a popular UI/Application testing tool developed by Google, on every app with an updated library for 2 hours. Although we did observe some crashes, we have confirmed that they are the bugs in the original apps. No new crash is introduced by LibBandAid. The results demonstrate that the updated library can function normally and pass the random testing successfully without any crash.

Due to the code coverage issue for random testing, we augment it with manual investigation to try out all the combinations of UI components. Combined with Monkey, our testing achieves an average code coverage of 25.7% for all the updated libraries. A closer look shows that our testing covers 30.1% of the functions that are actually updated. Admittedly, the code coverage is still far from satisfactory, however, the correctness of LibBandAid can still be demonstrated together with our manual investigation showed in the previous Section 4.7.2.

4.7.4 Effectiveness of *Value-sensitive Differential Slicing*

Finally, we evaluate the effectiveness of the new slicing algorithm by comparing it with the traditional algorithm. We seek to evaluate the algorithm by answering the two following questions:

1. How well does *Value-sensitive Differential Slicing* perform to reduce the over-conservativeness?

2. Can it help LibBandAid achieve better updating results?

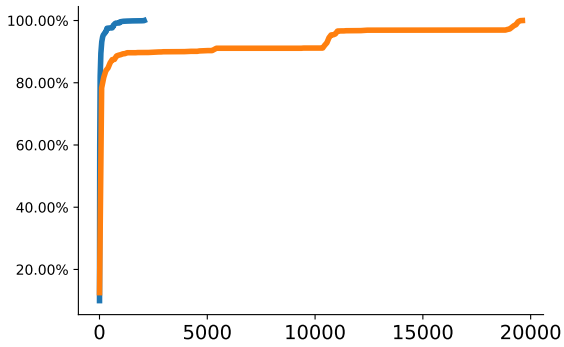


Figure 4.4: CDF for number of edges

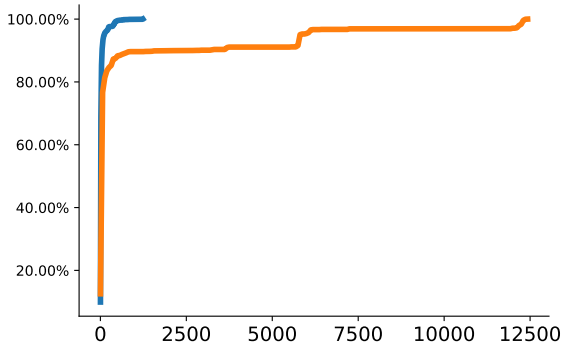


Figure 4.5: CDF for number of nodes

Figure 4.6: Effectiveness of New Slicing Algorithm

Over-conservativeness Reduction. We evaluate the effectiveness of *Value-sensitive Differential Slicing* by examining how well it could reduce the over-conservativeness across the 9 testing libraries. Figure 4.6 displays the cumulative distributions of the sizes of generated slices for traditional slicing as well as the new slicing with respect to the numbers of edges and nodes. The blue line indicates the new slicing algorithm while the yellow line represents the traditional slicing.

From the figures, we can see that *Value-sensitive Differential Slicing* could effectively reduce the number of edges as well as nodes by at least one order of magnitude. For example, 100% of the generated slices by *Value-sensitive Differential Slicing* have less than 2,500 edges and 2,000 nodes. On the contrary, traditional slicing generates way larger slices up to 20,000 edges and 12,500 nodes. This information gives us a clear view for the advantage of our slicing over the traditional slicing in terms of over-conservativeness reduction.

Updating Improvements. We further evaluate our algorithm by examining the updating results improvements. From the results shown in Section 4.7.2, LibBandAid could achieve a high successful updating rate for security commits when leveraging our new slicing algorithm. To evaluate, we simply run the experiments again with traditional slicing and compare the differences.

The results show that LibBandAid could only achieve a successful updating rate of 61.84% with a rate of 74.95% when combined with the potentially patchable commits. In contrast, with the help of *Value-sensitive Differential Slicing*, our system could perform much better at rates of 80.6% and 94.07%, respectively. Detailed information is presented in Figure 4.36.

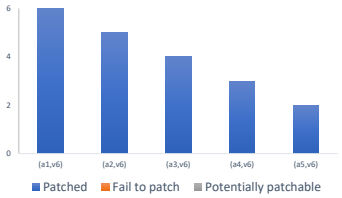


Figure 4.7: Butterknife

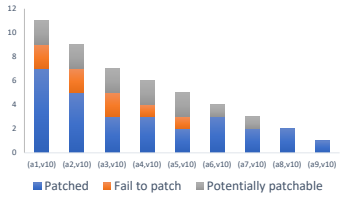


Figure 4.8: Dropbox

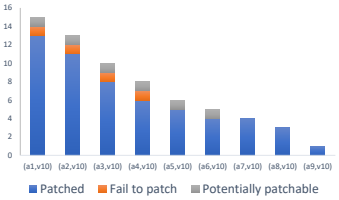


Figure 4.9: EventBus

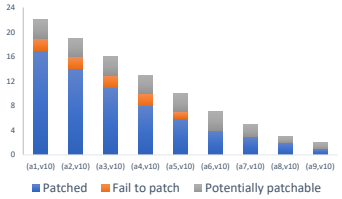


Figure 4.10: Glide

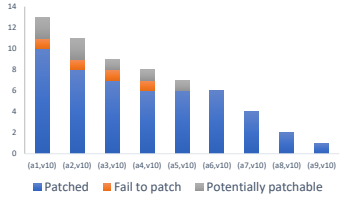


Figure 4.11: Gson

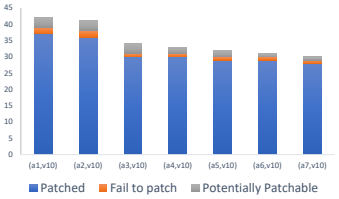


Figure 4.12: Leakcanary

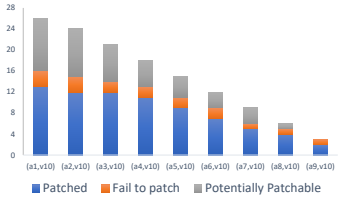


Figure 4.13: Okhttp

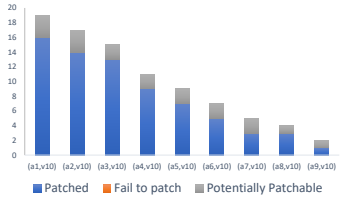


Figure 4.14: Picasso

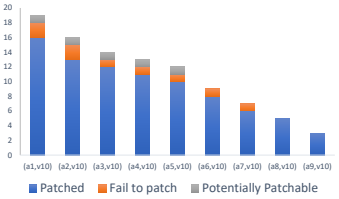


Figure 4.15: Retrofit

Figure 4.16: Effectiveness Results By Numbers

Table 4.4: Effectiveness Results By Vulnerability Category

Vulnerabilities	**Total**	**Failures**	**Failure Rate**
Race Condition	3	0	0%
Improper Access Control	2	0	0%
Uncontrolled Resource Consumption	1	0	0%
System Hang	5	0	0%
Illegal Reflective Access	1	0	0%
Stack Overflow	7	0	0%
Heap Access Error	3	0	0%
Missing Initialization	2	0	0%
Integer Overflow	1	0	0%
Fake ID	1	0	0%
New Security Feature	1	0	0%
Memory Leak	38	1	2.63%
Uncaught Exception	28	2	7.14%
Data Handling Error	36	3	8.33%
Uncheck Return Value	7	1	14.28%
Improper Input Validation	34	5	14.7%
Info Leak	3	1	33.33%

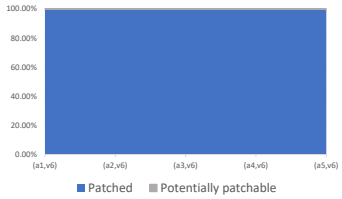


Figure 4.17: Butterknife

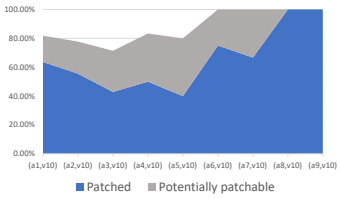


Figure 4.18: Dropbox

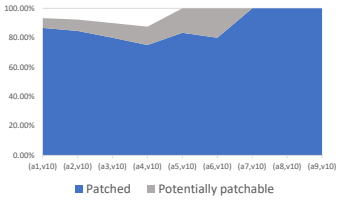


Figure 4.19: EventBus

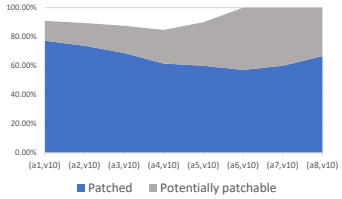


Figure 4.20: Glide

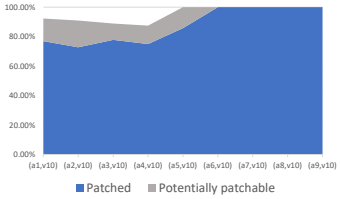


Figure 4.21: Gson

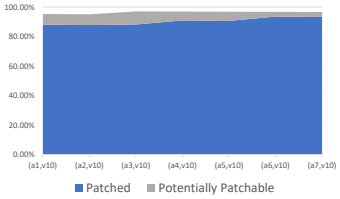


Figure 4.22: Leakcanary

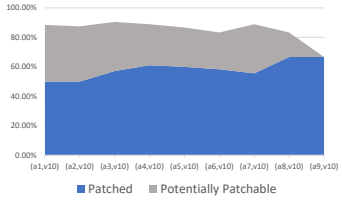


Figure 4.23: Okhttp

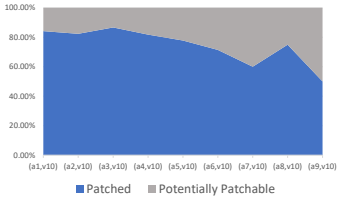


Figure 4.24: Picasso

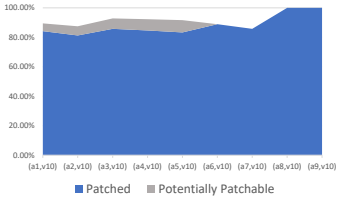


Figure 4.25: Retrofit

Figure 4.26: Effectiveness Results by Percentage

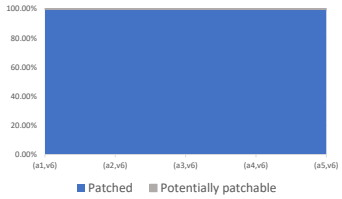


Figure 4.27: Butterknife

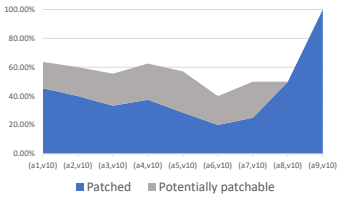


Figure 4.28: Dropbox

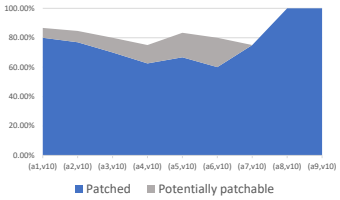


Figure 4.29: EventBus

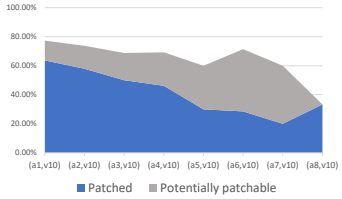


Figure 4.30: Glide

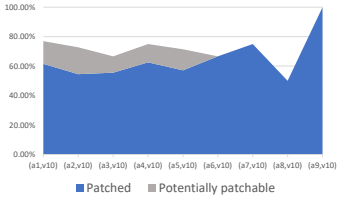


Figure 4.31: Gson

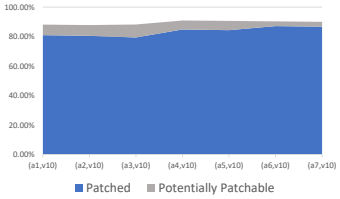


Figure 4.32: Leakcanary

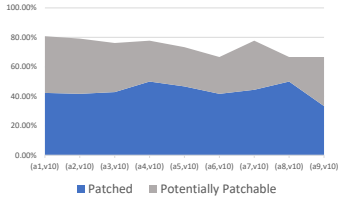


Figure 4.33: Okhttp

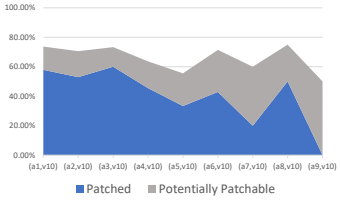


Figure 4.34: Picasso

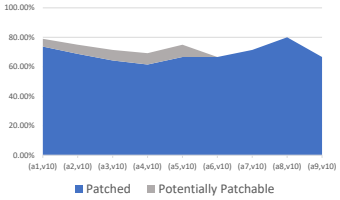


Figure 4.35: Retrofit

Figure 4.36: Effectiveness Results with Traditional Slicing

Chapter 5

Learning Program-Wide Code Representations for Binary Diffing

5.1 Introduction

Binary Code Differential Analysis, a.k.a, binary diffing, is a fundamental analysis capability, which aims to quantitatively measure the similarity between two given binaries and produce the fine-grained block level matching. Given two input binaries, it precisely characterizes the program-wide differences by generating the optimal matching among the blocks with quantitative similarity scores. It can not only present a more precise, fine-grained and quantitative results about the differences at a whole binary scale but also explicitly reveal how code evolves across different versions or optimization levels. Due to this level of precision and granularity, it has enabled many critical security usages in different scenarios when program-wide analysis is required, such as changed parts locating [28], malware analysis [66,

101], patch analysis [132, 89], binary wide plagiarism detection [94] and patch-based exploit generation [40].

Because of the importance, binary diffing has been an active research focus. BinDiff [34] which is the state-of-the-art commercial binary diffing tool performs many-to-many graph isomorphism detection on callgraph and control-flow graph (CFG) and leverages heuristics (e.g., function name hash, graph edge MD index) to match functions as well as blocks. Other techniques perform diffing on the generated flow graphs [60, 108, 65] or decompose the binaries into fragments [57, 55, 56] for similarity detection. These approaches do not consider the semantics of instructions which can be critical during analysis, especially when dealing with different compiler optimization levels. Moreover, traditional graph matching algorithm such as Hungarian algorithm [83] is expensive and cannot guarantee optimal matching.

Another line of research utilizes dynamic analysis for diffing. These techniques carry out the analysis by directly executing the given code [61, 126], performing dynamic slicing [100] or using symbolic execution [70, 99, 94] on the given binaries and checking the semantic level equivalence based on the information collected during the execution. In general, these techniques excel at extracting semantics of the code and have good resilience against compiler optimizations and code obfuscation but usually suffer from very poor scalability and incomplete code coverage because of the nature of dynamic analysis.

Recently, researchers have been leveraging the advance of machine learning to tackle the problem. Various techniques such as Genius [68], Gemini [131], INNEREYE [150] and Asm2Vec [59] have been proposed to utilize graph representation learning techniques [53,

98, 86] and incorporate code information into embeddings (i.e, high dimensional numerical vectors). Then they use these embeddings for similarity detection. INNEREYE [150] and Asm2Vec [59] further leverage NLP techniques to automatically extract semantic information and generate embeddings for diffing.

These approaches embrace multiple advantages over the traditional static and dynamic approaches: 1) higher accuracy as they incorporate unique features of the code into the analysis by using either manual engineered features [68, 131] or deep learning based automatic methods [59, 150]; 2) better scalability since they avoid heavy graph matching algorithm or dynamic execution. Moreover, the deep learning process can be significantly accelerated by GPUs.

Limitations to Existing Learning based Techniques. Despite the advantages, we identify three major limitations for the existing learning based approaches.

First, no existing technique can perform program-wide binary diffing at a fine-grained basic block level. They either perform diffing on functions [68, 131, 84, 59] or on small code components [150], and tells how two given functions or small pieces of blocks are similar. As mentioned above, fine-grained binary diffing is an important analysis upon which many critical security analysis can be built. Hence, a more fine-grained binary diffing tool is strongly desired.

Second, none of them considers both program-wide dependency information as well as basic block semantic information during analysis. INNEREYE [150] extracts block semantic information with NLP techniques [98] but only considers local control dependency information within a small code component by adopting the Longest Common Subsequence

(LCS). Asm2Vec [59] generates multiple random walks within functions and uses the walks to learn token and function embeddings. It is especially troublesome when performing binary diffing as one binary could contain multiple very similar functions. In this case, program-wide function calling relations can be vital in differentiating these functions.

Third, most of the learning based techniques [132, 84, 150] are based on supervised learning. Thus, the performance is heavily dependent on the quality of training data. We argue that a large, representative and balanced training dataset can be very hard to collect in binary diffing problem due to the extreme diversity of the binary programs.

Our Approach. To this end, we propose an unsupervised deep neural network (DNN) based program-wide code representation learning technique for binary diffing. In particular, our technique learns basic block level embeddings for binary diffing via completely unsupervised learning. Each learned embedding represents a specific block by carrying not only the semantic information of the block but also the structural information from the program-wide inter-procedural control flow graph (ICFG). These embeddings are used to efficiently and accurately calculate the similarities.

To achieve this, we leverage NLP techniques to generate token (opcode and operands) embeddings which are further averaged and concatenated to assemble block level feature vectors. These feature vectors contain the semantic information for specific blocks. Then, we merge the two ICFGs of the input binaries on selected terminal nodes (no outgoing edge) so that the merged graph contains program-wide structural information for both binaries yet the original dependency information remains unchanged. Matrix factorization is then performed on the graph along with the generated feature vectors to produce block level

embeddings using Text-associated DeepWalk algorithm (TADW) [134]. Consequently, these block embeddings contain both the program-level structural information as well as the semantics from the blocks. Finally, to deal with the unique challenge of binary diffing, we present a K-hop greedy matching algorithm to match blocks and confront compiler optimizations including function inlining, instruction reordering, etc.

We implement a prototype named DeepBinDiff and conduct an extensive evaluation with three representative datasets containing 113 binaries. The evaluation results show that our tool outperforms the state-of-the-art tools BinDiff and Asm2Vec by large margin in terms of effectiveness with respect to cross-version and cross-optimization level diffing. Furthermore, we also conduct a case study using real-world vulnerabilities in OpenSSL [33]. The case study also shows that our tool has unique advantages when analyzing vulnerabilities. To our best knowledge, our evaluation is the only work that comprehensively examines the cross-version and cross-optimization binary diffing problem.

5.2 Problem Statement

In this section, we formalize the problem definition for binary diffing problem and further describe our problem statement and design goals.

5.2.1 Problem Definition

Given two binary programs, binary diffing precisely measures the similarity and characterizes the differences between the two binaries at a fine-grained basic block level. We formally define ***binary diffing problem*** as follows:

Definition 4. Given two binary programs $p_1 = (B_1, E_1)$ and $p_2 = (B_2, E_2)$, binary diffing aims to find the optimal block matching that maximizes the similarity between p_1 and p_2:

$$SIM(p1, p2) = \max_{m_1, m_2, ..., m_k \in M(p_1, p_2)} \sum_{i=1}^{k} sim(m_i), \text{ where:}$$

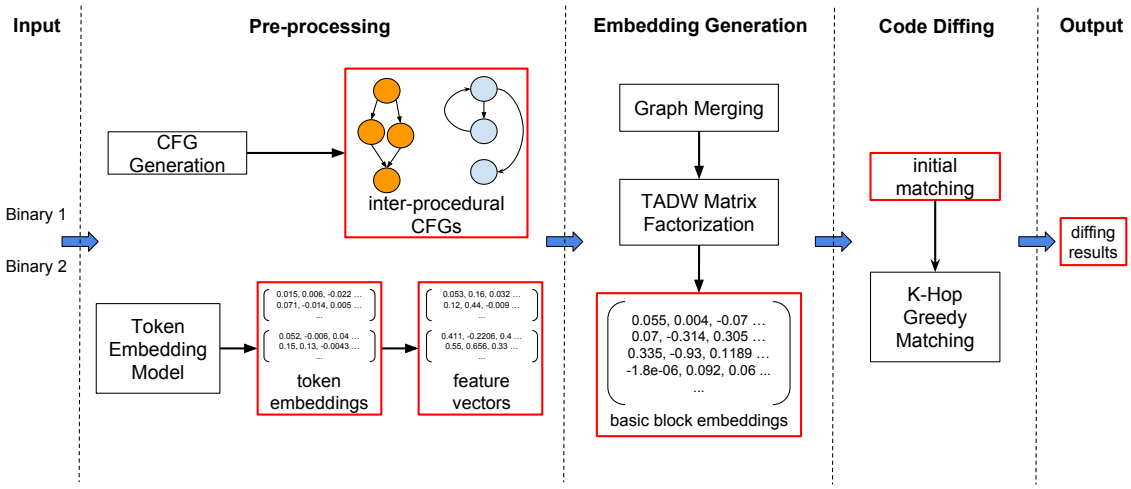
- $B_1 = \{b_1, b_2, ..., b_n\}$ and $B_2 = \{b_1^{'}, b_2^{'}, ..., b_n^{'}\}$ are two sets containing all the basic blocks in p_1 and p_2;
- Each element e in $E \subseteq B \times B$ corresponds to *control flow dependency* between two basic blocks;
- Each element m_i in $M(p_1, p_2)$ represents a matching pair between b_i and $b_i^{'}$;
- $sim(m_i)$ defines the quantitative similarity score between two matching basic blocks.

Therefore, the problem can be transformed into two subtasks: 1) introducing $sim(m_i)$ which quantitatively measures the similarity between two basic blocks; 2) finding the optimal matching between two sets of basic blocks $M(p1, p2)$.

5.2.2 Assumptions

To formalize our problem domain, we list the following assumptions on the given inputs:

- Only stripped binaries are given. This assumption in general makes the problem harder as no source or symbol information is presented. However, this assumption is very realistic as COTS binaries are often stripped and malware binaries do not carry internal symbols for obvious reasons.



- Binaries are not packed but can be optimized with different compiler optimization levels. Different optimization levels can result in distinctive binary codes even with the same source code input. Tolerating differences in binaries introduced by optimization levels is also very important to the code diffing problem. From the evaluation we can see that even the state-of-the-art tools cannot handle them well. So for packed malware binaries, we assume they are unpacked by using existing unpacking tools before being present to our tool.

- Two input binaries are for the same architecture. So far DeepBinDiff supports x86 binaries since they are the most prevalent in real world binaries. Of course, DeepBinDiff could be extended to handle cross-architecture diffing by performing analysis on an Intermediate Representation (IR) level. We leave it as future work.

5.3 Approach Overview

Figure 5.3 delineates the system architecture of DeepBinDiff. Red squares in the figure represent generated intermediate data during analysis. As shown, the system takes as input two binaries and outputs the basic block level binary diffing results. The system solves the two tasks mentioned in Section 5.2.1 by using two major techniques. First, to introduce $sim(m_i)$ which quantitatively measures the similarity between two blocks, DeepBinDiff embraces an unsupervised deep learning approach to generate embeddings and utilizes embeddings to efficiently calculate the similarity scores between blocks. Second, a K-hop greedy matching algorithm is executed to generate the matching $M(p_1, p_2)$.

The whole system consists of three major components: 1) pre-processing; 2) embedding generation and 3) code diffing. Pre-processing which can be further divided into two subcomponents: CFG generation and feature vector generation, is responsible for generating two pieces of information: inter-procedural control-flow graphs (ICFGs) and feature vectors for basic blocks. Once generated, the two results are sent to embedding generation component which utilizes TADW technique [107] to learn the graph embeddings for each node (basic block). DeepBinDiff then makes use of the generated block embeddings and leverages a K-hop greedy matching algorithm to perform code diffing at basic block level.

5.4 Pre-processing

Pre-processing takes the input binaries, analyzes them and produces inputs for embedding generation component. More specifically, it uses IDA pro [32] to generate inter-procedural CFGs for the input binaries and applies a token embedding generation model

to generate embeddings for each token (opcode and operands). The model is trained using binaries by leveraging a popular deep learning algorithm Word2Vec [98]. These generated token level embeddings are further transformed into basic block level feature vectors.

5.4.1 CFG Generation

By combining call graph of the binary with control-flow graphs of each function, inter-procedural CFG (ICFG) contains control dependency information among basic blocks cross function boundaries. ICFG in DeepBinDiff plays a very important role, that is to provide program-wide contextual information when calculating block similarities. This information is particularly useful when there exist multiple semantically similar blocks. In this case, this contextual information can be of great help in differentiating them. However, none of the existing techniques assimilate this information for binary diffing. DeepBinDiff leverages IDA pro [32] to extract basic block information and generates inter-procedural CFGs for the two given binaries.

5.4.2 Feature Vector Generation

Besides the structural control dependency information carried by ICFGs, DeepBinDiff also takes into account the semantic information from basic blocks by generating feature vector for each block.

Existing techniques such as Genius [68] and Gemini [131] empirically select some features from basic blocks and control flow graphs and then embed them into the CFGs to form attributed CFGs (ACFG). However, by manually selecting limited number of features, one could easily miss some essential information and impose bias to the results. To overcome

this limitation, INNEREYE [150] takes a supervised learning approach, utilizes Word2Vec to generate instruction embeddings and further deploys an LSTM-RNN model to convert instruction embeddings to block embeddings. It requires sizable and balanced training dataset with labels (which can be hard to extract) to train the LSTM model. Asm2Vec [59], on the other hand, uses an unsupervised PV-DM model to produce the token and function embeddings. In DeepBinDiff, we take the unsupervised learning approach as well due to the fact that labels and balanced training dataset can be very hard to generate in binary diffing scenario.

In DeepBinDiff, we leverage an unsupervised NLP technique Word2Vec [98] model which can generate embeddings for each word based on its context (words around it) to extract the semantics of each block. In our case, we consider each token (opcode or operand) as word, generate random walks on top of ICFGs to be sentences and instructions around each token as its context.

The whole process consists of two subtasks: token embedding generation and feature vector generation. More specifically, we first train a token embedding model using Word2Vec, then use this model to generate token embeddings. These token embeddings are further averaged and concatenated to generate feature vectors for blocks. Hence, the major component is the token embedding model generation depicted in Figure 5.1. It takes 3 steps: 1) random walk; 2) normalization and 3) model training.

Random Walks When distilling semantics of each token, we would like to make use of the instructions around it as its context. Therefore, we need to serialize ICFGs to extract

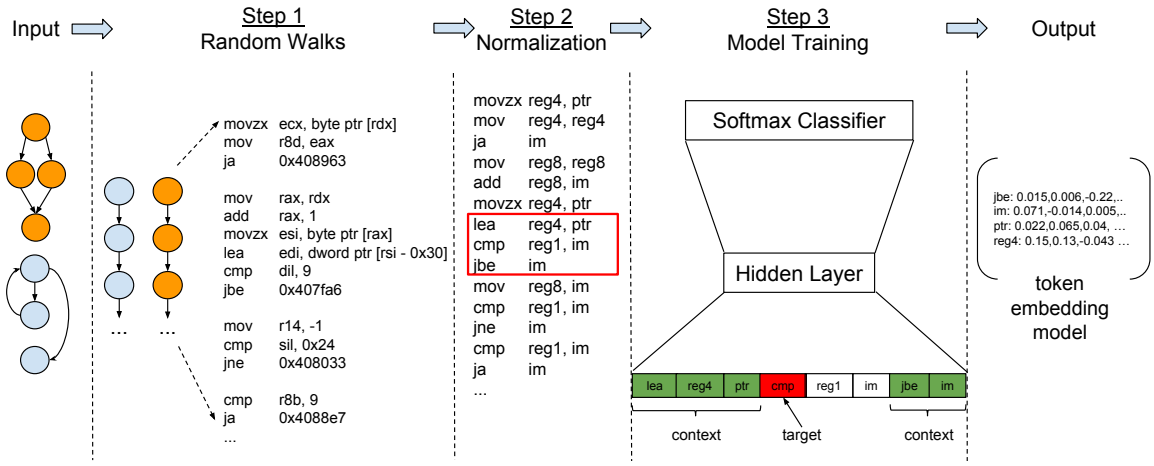


Figure 5.1: Token Embedding Model Generation.

control flow dependency information. As shown in Step 1 in Figure 5.1, we do this by generating random walks for each node in ICFGs so that each random walk contains one possible execution path of the binary. We then consider these random walks as sentence for Word2Vec algorithm. Empirically, we generate 2 random walks per block to make sure every block is covered and each random walk has a length of 5 blocks to ensure enough control flow information is carried. Then, we put random walks together to generate a complete article for training.

Normalization Before sending the article to train our Word2Vec model, the serialized codes may still contain some differences due to various compilation choices. To refine the code, DeepBinDiff adopts a code normalization process.

Shown as Step 2 in Figure 5.1, our system conducts the normalization using the following rules : 1) all numeric constant values are replaced with string 'im'; 2) all general registers are renamed according to their lengths; 3) pointers are replaced with string 'ptr'. It is noteworthy that we do not follow INNEREYE [150] where all the string literals are

replaced with $< STR >$. This is because we believe the string literals are very useful to distinguish different blocks.

Model Training Then DeepBinDiff applies the popular Word2Vec algorithm[98] to the generated article in order to learn the token embeddings.

Word2Vec Algorithm A word embedding is simply a vector which is learned from the given articles to capture the contextual semantic meaning of the word. There exist multiple methods to generate vector representations of words including the most popular Continuous Bag-of-Words model (CBOW) and Skip-Gram model proposed by Mikolov et al. [98]. Here we utilize the CBOW model which predicates target from its context.

$$J(w) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c}\log p(w_{t+j}|w_t) \tag{5.1}$$

Given a sequence of training words $w_1, w_2, ..., w_t$, the objective of the model is to maximize the average log probability $J(w)$ as shown in Equation 5.1

where c is the sliding window for context and $p(w_{t+j}|w_t)$ is the softmax function defined as Equation 5.2.

$$p(w_k \in C_t|w_t) = \frac{\exp(v_{w_t}^T v_{w_k})}{\sum_{w_i \in C_t}\exp(v_{w_t}^T v_{w_i})} \tag{5.2}$$

where v_{w_t}, v_{w_k} and v_{w_i} are the vector representations of w_t, w_k and w_i. To further improve the efficiency of the computation, Word2Vec adopts the hierarchical softmax as a computationally efficient approximation [98].

Token Embeddings To train the token embedding generation model shown as Step 3 in Figure 5.1, we modify the Word2Vec CBOW model that uses words around a target word as context. In our case, tokens (opcode and operands) are the words. However, we consider instructions instead of tokens around the target token as context, shown in Step 3 in Figure 5.1. For example, in the figure, the current target token is *cmp* (shown in red), so we use one instruction before and another instruction after (shown in green) in the random walk as the context.

Our token embedding generation model is inspired by Asm2Vec which also uses instructions around a target token as context. Nonetheless, our model has a fundamentally different design goal than Asm2Vec. DeepBinDiff learns token embeddings via program-wide random walks while Asm2Vec is trying to learn function and token embeddings at the same time and only within the function. Therefore, we choose to modify Word2Vec CBOW model while Asm2Vec leverages the PV-DM model so that it can generate the two embeddings at the same time.

Feature Vector Generation

Once the token embeddings are generated, the last step is to generate feature vectors for basic blocks. Since each basic block could contain multiple instructions, each instruction in turn involves one opcode and potentially multiple operands, we calculate the average of the operand embeddings and then concatenate with the opcode embedding.

Particularly, for an instruction in_i which contains an opcode p_i and a set of k (could be zero) operands Set_{t_i}, we model the instruction embedding as concatenation of opcode embedding and the average of operand embeddings. Therefore, for a block $b = \{in_1, in_2, .., in_j\}$

which contains j instructions, its feature vector FV_b is the sum of its instruction embeddings, as depicted in Equation 5.3.

$$FV_b = \sum_{i=1}^{j}(embed_{p_i}||\frac{1}{|Set_{t_i}|} * \sum_{n=1}^{k} embed_{t_{i_n}}) \tag{5.3}$$

5.5 Embedding Generation

Based on the ICFGs and feature vectors generated in prior steps, this component produces embeddings for basic blocks in the two input binaries with a goal that similar blocks can be associated with similar embeddings. Hence, block embeddings could be leveraged for binary diffing in later steps. To do so, DeepBinDiff first merges the two ICFGs into one graph and then perform matrix factorization based on Text-associated DeepWalk algorithm (TADW) [134] to generate block embeddings.

Since the most important building block for this component is the TADW algorithm, we first describe the algorithm in detail and present how basic block embeddings are generated. Then, we justify why graph merging is needed for TADW and report how DeepBinDiff accomplishes it.

5.5.1 TADW algorithm

Text-associated DeepWalk algorithm [134] is an automatic graph embedding learning technique based on unsupervised deep learning. As suggested by name, it can be considered as an improvement over the DeepWalk algorithm [107].

DeepWalk DeepWalk algorithm is an online graph embedding learning algorithm that considers random walks as a corpus in language modeling problem, and the graph vertices as its own vocabulary. The embeddings are then learned using random walks on the vertices in the graph. Accordingly, vertices that share similar neighbours will have similar embeddings.

More specifically, given a graph $G = (V, E)$ where V represents all the vertices and E contains all the edges, the algorithm gets a shuffle of V and generates *random walks* with a pre-defined length t for each vertex in the shuffle and applies Skip-Gram model as defined previously in Equation 5.2 with respect to a fixed window size w. Just like Word2Vec, it also adopts the hierarchical Softmax to improve the efficiency.

Text-associated DeepWalk As described, DeepWalk excels at learning the structural information from a graph. Nevertheless, it does not consider the features from each vertex and differentiate them by their own uniqueness. As a result, Yang at el. [134] propose an improved algorithm called Text-associated DeepWalk (TADW) which is able to incorporate features of vertices into the network representation learning process.

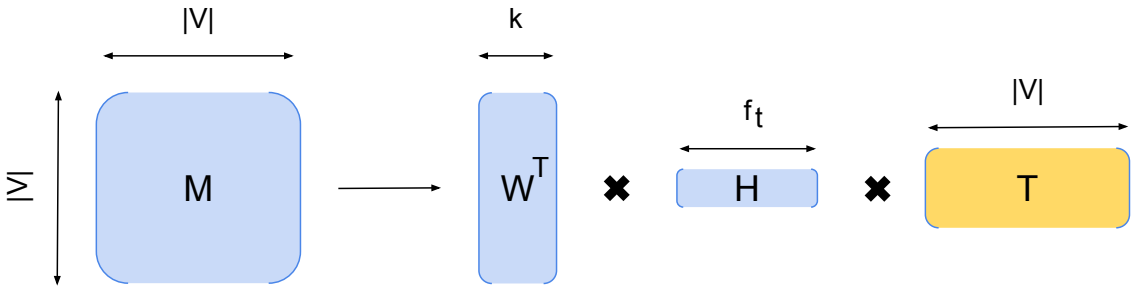


Figure 5.2: TADW

It is proven that DeepWalk is equivalent to factorizing a matrix $M \in R^{|v| \times |v|}$ where each entry M_{ij} is logarithm of the average probability that vertex v_i randomly walks

to vertex v_j in fixed steps. This discovery further leads to TADW algorithm depicted in Figure 5.2. It shows that it is possible to factorize the matrix M into the product of three matrices: $W \in R^{k \times |v|}$, $H \in R^{k \times f}$ and a text feature $T \in R^{f \times |v|}$. Then, W is concatenated with HT to produce $2k$-dimensional representations of vertices (embeddings).

5.5.2 Graph Merging

The goal for embedding generation of DeepBinDiff is to learn block embeddings such that similar blocks correspond to similar embeddings. To achieve this, DeepBinDiff leverages TADW to generate embedding for each block. Since we have two ICFGs (one for each binary), the most intuitive way is to run TADW twice for the two graphs, hence, blocks that hold similar semantic (feature vectors) and structural information can obtain similar embeddings. However, this method has two drawbacks. First, it is inefficient to perform matrix factorization twice. Second, generating embeddings for the two graphs individually could in fact lower the effectiveness.

Take the example exhibited in Figure 5.3 for illustration. In this figure, we have two ICFGs and each graph has one block that calls a libc function *fread* and another block that has a reference to string *'hello'*. Ideally, there is a great chance that these two pairs of nodes ('a' and '1', 'd' and '3') should matched. However, the feature vectors of these blocks may not be very similar as one block could contain multiple instructions and call or reference instruction is just one of them. Also, the two pairs also have quite different structural information. For example, node 'a' has two outgoing edges but no incoming edges while node '1' has. As a result, it is possible that DeepBinDiff cannot generate very similar embeddings for the two pairs of nodes.

To alleviate the problem and make TADW run only once, DeepBinDiff adopts a graph merging process to merge the two ICFGs before TADW. Particularly, it leverages IDA Pro to extract the string references and detect calls to external libraries and system calls. Then, DeepBinDiff creates new virtual nodes for the strings and external library functions and draws edges from the callsites to these virtual nodes. This way, two graphs are merged into one graph on some terminal virtual nodes. By doing so, node 'a' and '1' will have at least one common neighbor which boosts the similarity between them. Also, since we only merge the graphs on terminal nodes, the original structures of the two graphs are unchanged.

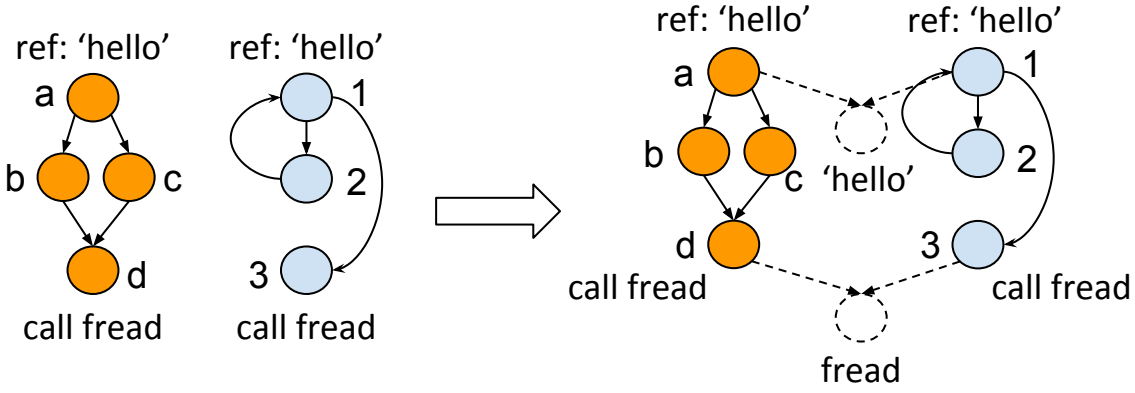


Figure 5.3: Graph Merging

5.5.3 Basic Block Embeddings

With the merged graph, DeepBinDiff leverages TADW algorithm and performs matrix factorization to generate basic block embeddings.

More specifically, DeepBinDiff feeds the merged graph from two ICFGs and the block feature vectors into TADW for multiple iterations of optimization. The algorithm

factorizes the matrix M into three matrices by minimizing the loss function depicted in Equation 5.4 using Alternating Least Squares (ALS) algorithm [81]. It stops when the loss converges or after a fixed n iterations.

$$\min_{W,H} ||M - W^T HT||_F^2 + \frac{\lambda}{2}(||W||_F^2 + ||H||_F^2) \tag{5.4}$$

On that account, each generated basic block embedding contains not only the information about the block itself, but also the information from the ICFG structural information. The generated embeddings are essential for binary diffing.

5.6 Code Diffing

Once the basic block embeddings are generated, the last step in DeepBinDiff is to perform code diffing. The goal is to find the optimal matching between blocks that maximizes the similarity for the two input binaries.

One spontaneous choice for this task is to consider blocks from one binary as workers, blocks from the other binary as jobs and embedding similarities as weights, then perform linear assignment to come up with the global optimal matching. This method, however, suffers from two major limitations. First, binaries could contain thousands of blocks or even more and linear assignment at this scale is inefficient. Second, although embeddings do include structural information, linear assignment itself does not consider any graph information. Thus, it is still likely to make mistakes when matching very similar blocks. Another possible way to improve the performance is to conduct two-level linear assignment at function level and block level. Instead of matching blocks directly, we match

functions first by using function level embeddings. Then, blocks within the matched function pairs can be further matched with block embeddings. This approach can be thwarted by compiler optimizations that alter the function boundary such as function inlining.

5.6.1 K-Hop Greedy Matching

To tackle this problem, we introduce a K-hop greedy matching algorithm. The idea is to benefit from the ICFG structural information and find matching blocks based on the similarity calculated from embeddings within the neighbors of already matched ones.

As presented in Algorithm 4, it extracts initial matching $Set_{initial}$ from the inserted virtual nodes during graph merging described in Section 5.5.2 and produces $Set_{matched}$ as the binary diffing result. The initial matched pairs are generated by using the string references and calls to external functions. For example, if each binary only has one block that refers to string *"hello world"*, it is highly likely that these two blocks will match. Starting from the initial set, the algorithm loops and explores the neighbors of the already matched pairs in $GetKHopNeighbors()$ in Ln.7-8. DeepBinDiff then sorts the similarities between neighbor blocks and picks the pair bearing highest similarity by calling $FindMaxUnmatched()$ in Ln.9. During this step, the algorithm empirically sets a threshold of 0.25 to make sure the new matching blocks are similar enough. This process is repeated until all K-neighbors of matched pairs are explored and matched. Note that after the loop, there may still exist unmatched blocks due to unreached code (dead code) or low similarity within K-hop neighbors. The algorithm then performs linear assignment and finds the optimal matching among them in Ln.16. Finally, the algorithm returns $Set_{matched}$ as the binary diffing result.

Algorithm 4 K-Hop Greedy Matching Algorithm

```
1: Set_initial ← {initial match from virtual nodes}
2: Set_matched ← Set_initial
3: Set_currPairs ← Set_initial
4:
5: while Set_currPairs != empty do
6:     (node_1, node_2) ← Set_currPairs.pop()
7:     nb_node1 ← GetKHopNeighbors(node_1)
8:     nb_node2 ← GetKHopNeighbors(node_2)
9:     newPair ← FindMaxUnmatched(nb_node1, nb_node2)
10:    if newPair != NULL then
11:        Set_matched ← Set_matched ∪ newPair
12:        Set_currPairs ← Set_currPairs ∪ newPair
13:    end if
14: end while
15: Set_unreached ← {blocks that are not yet matched}
16: Set_matched ← Set_matched∪ LinearAssign(Set_unreached)
    output Set_matched as the binary diffing result
```

5.7 Evaluation

In this section, we evaluate DeepBinDiff with respect to its effectiveness and efficiency for two different binary diffing scenarios: cross-version diffing and cross-optimization level diffing. To our best knowledge, this is the first research work that comprehensively examines the effectiveness of binary diffing tools under the cross-version setting.

Furthermore, we conduct a case study to demonstrate the usefulness of DeepBinDiff in real-world vulnerability analysis.

5.7.1 Experimental Setup

Our experiments are performed on a moderate desktop computer running Ubuntu 18.04LTS operating system with Intel Core i7 CPU, 16GB memory and no GPU. The feature vector generation and block embedding generation components in DeepBinDiff

are expected to be significantly accelerated if GPUs are utilized since the two components are built upon deep learning models.

5.7.2 Datasets & Baseline Techniques

Datasets. To thoroughly evaluate the effectiveness of DeepBinDiff, we collect a number of representative datasets. More specifically, we utilize three binary sets - Coreutils [29], Diffutils [30] and Findutils [31] with a total of 113 binaries for the evaluation of effectiveness and efficiency. Multiple different versions of the binaries (5 versions for Coreutils, 4 versions for Diffutils and 3 versions of Findutils) are collected with wide time spans between the oldest and newest versions (13, 15, 7 years respectively). This setting ensures that each collected version has enough distinctions such that binary diffing results among them are meaningful and representative.

We then compile the programs using GCC 5.4 with 3 different compiler optimization levels (O1, O2 and O3) to produce binaries equipped with different optimization techniques. This dataset is leveraged to show the effectiveness of DeepBinDiff in terms of cross-optimization level diffing.

Moreover, we leverage a popular general-purpose cryptography library OpenSSL [33] for vulnerability analysis case study. In the case study, we use two different real-world vulnerabilities to demonstrate the advantage of DeepBinDiff in practice.

Baseline Techniques. With the aforementioned datasets, we run DeepBinDiff and compare it against another two state-of-the-art baseline techniques (Asm2Vec [59] and BinDiff [34]). Note that Asm2Vec is designed only for function level similarity detection. We

leverage its algorithm to generate token embeddings and take the same procedure in the paper to average operand embeddings and concatenate with opcode embedding to generate instruction embeddings. Then, we further sum them up to produce block embeddings and perform binary diffing with the same K-hop greedy matching algorithm adopted by DeepBinDiff.

5.7.3 Ground Truth Collection

Ground truth information about how blocks from two binaries should be matched is required when measuring the effectiveness of DeepBinDiff. To this end, we rely on source code matching and debug symbol information to collect ground truth information.

Particularly, for two input binaries to be diff'ed, we first extract source file names from the binaries and then use Myers algorithm [103] to perform text based matching for the source code of the two binaries in order to get the line number matching. We take two steps to ensure the soundness of our extracted ground truth information. First, we only collect identical lines of source code as matching but ignore the modified ones. Second, our ground truth collection process is being deliberately conservative to remove the matching lines like macros which could expand to multiple lines of code. Therefore, although our source code matching is by no means complete, it is guaranteed to contain zero false positive.

Once we have the line number mapping between the two binaries, *readelf* tool is used to extract debug info to understand the mapping between line numbers and program addresses. Eventually, the ground truth is collected by examining the blocks of the two binaries containing program addresses that map to the matched line numbers. This way, we can successfully collect ground truth block matching information for every pair of binaries.

Example. Take the diffing between v5.93 and v8.30 of Coreutils binary *chown* for illustration. To collect ground truth information for block matching between the two binaries, we first extract source file names from them and perform text-based matching between the corresponding source files. By matching the source files chown.c in the two versions, we know Ln. 288 in v5.93 should be matched to Ln. 273 in v8.30. Together with the debug information extracted by *readelf*, a matching between address 0x401cf8 in v5.93 *chown* and address 0x4023fc in v5.93 *chown* can be established.

Finally, we use IDA Pro to generate basic blocks for the two binaries. By checking the addresses within the blocks, we know block 3 in v5.93 *chown* should be matched to block 13 in v8.30 *chown*. Therefore, we collect the ground truth information about how the blocks between the two binaries should be matched and we further use this information to measure the correctness of the diffing results produced by DeepBinDiff.

5.7.4 Effectiveness

With the experimental datasets and ground truth information collected, we evaluate the effectiveness of DeepBinDiff by performing diffing between binaries across different versions and optimization levels, and checking the results against the ground truth information. We also compare the results against two state-of-the-art baseline techniques.

Evaluation Metrics In the evaluation, we use precision and recall metrics to measure the effectiveness of the diffing results produced by diffing tools. The matching result M from DeepBinDiff for two given binaries can be presented as a set of block matching pairs with a length of x. Similarly, the ground truth information G for the two binaries can be presented

as a set of block matching pairs with a length of y. We present them in Equation 5.5 and 5.6. Elements in set M and G show matching relationship between two basic blocks from the two binaries in the matching result from DeepBinDiff and ground truth information respectively.

$$M = \{(m_1, m_1^{'}), (m_2, m_2^{'}), ..., (m_x, m_x^{'})\} \tag{5.5}$$

$$G = \{(g_1, g_1^{'}), (g_2, g_2^{'}), ..., (g_y, g_y^{'})\} \tag{5.6}$$

We then define two subsets of M: M_c and M_u, representing correct matching and unknown matching respectively. Correct match $M_c = M \cap G$ is the intersection of our result M and ground truth G which gives us the correct block matching pairs. Unknown matching result M_u represents the block matching pairs that no block in these pairs is ever appeared in ground truth. Thus, we have no idea whether these matching pairs are correct. This could happen due to the conservativeness of our ground truth collection process. Consequently, $M - M_u - M_c$ portrays the matching pairs in M that are not in M_c nor in M_u, therefore, all pairs in $M - M_u - M_c$ are confirmed to be incorrect matching pairs.

Once we have M and G formally presented, we use precision and recall metrics to show the quality of diffing results. The precision metric presented in Equation 5.7 gives us the percentage of correct matching pairs among all the known pairs (correct and incorrect).

$$Precision = \frac{||M \cap G||}{||M \cap G|| + ||M - M_u - M_c||} \tag{5.7}$$

The recall metric shown in Equation 5.8 is produced by finding the intersection of sets M and G and dividing its size by the size of set G. This metric shows the percentage of ground truth pairs that are correctly matched by the diffing tool.

$$Recall = \frac{||M \cap G||}{||G||} \tag{5.8}$$

Cross-version Diffing In this experiment, we benchmark the performance of DeepBinDiff, BinDiff and Asm2Vec by conducting binary diffing between different versions of binaries (all compiled with O1 compiler optimization level) in Coreutils, Diffutils and Findutils. We report the average recall and precision for each tool under different experimental settings in Table 5.1.

As we can see, DeepBinDiff outperforms Asm2Vec and BinDiff across all versions of the three datasets in terms of recall, especially when the two diffed versions have a large gap. For example, for Coretuils diffing between v5.93 and v8.30, DeepBinDiff improves the recall by 14% and 42% over Asm2Vec and BinDiff. Also, we can observe that Asm2Vec which carries the semantic information for tokens, in general has better recall than the de-facto commercial tool BinDiff. This evaluation results show that including semantic information during analysis can improve the effectiveness of diffing. Moreover, since a major difference between DeepBinDiff and Asm2Vec is the structural information generated from TADW, we can also draw the conclusion that structural information can help boost the quality of diffing results by a large margin.

Interestingly, we also notice that BinDiff sometimes can produce higher precision than Asm2Vec and DeepBinDiff. We investigate the detailed results and see that BinDiff

has a very conservative matching strategy. It usually only matches the basic blocks with very high similarity score and leaves the other blocks unmatched. Therefore, BinDiff generates much short matching list than DeepBinDiff which uses K-hop greedy matching to maximize the matching.

We further present the Cumulative Distribution Function (CDF) figures for F1-scores of the three diffing techniques on Coreutils binaries in Figure 5.8. Again, from the CDF figures we can see that Asm2Vec and BinDiff have similar F1-scores while DeepBinDiff performs much better. And due to the high precision, BinDiff can even have higher F1-scores than Asm2Vec. In a nutshell, DeepBinDiff can exceed two baseline techniques by large margins with respect to cross-version binary diffing.

Cross-optimization level Diffing We then conduct experiments to measure the effectiveness of the three techniques in terms of cross-optimization level binary diffing. We perform diffing between the binaries with the same version but under different optimization levels. Particularly, each binary is diffed twice (O1 versus O3 and O2 versus O3). We report the average recall and precision for all settings in Table 5.2.

As shown, just like cross-version binary diffing, DeepBinDiff could outperform Asm2Vec and BinDiff for most of the settings in recall rate. The only exception is the Diffutils v3.6 O1 to O3 diffing where Asm2Vec has a recall rate of 0.876 while DeepBinDiff obtains 0.865. It is because there are only 4 binaries in Difftuils and most of them are small. In this special case, program-wide structure information may become less useful. Still, DeepBinDiff could defeat Asm2Vec for all other settings, even for Diffutils. Also, BinDiff

Table 5.1: Cross-version Binary Diffing Results

		Recall			Precision		
		BinDiff	Asm2Vec	DeepBinDiff	BinDiff	Asm2Vec	DeepBinDiff
Coreutils	v5.93 - v8.30	0.506	0.633	**0.721**	**0.775**	0.611	0.713
	v6.4 - v8.30	0.572	0.664	**0.747**	**0.784**	0.641	0.745
	v7.6 - v8.30	0.748	0.777	**0.878**	0.771	0.753	**0.874**
	v8.1 - v8.30	0.756	0.792	**0.884**	0.821	0.766	**0.878**
Diffutils	v2.8 - v3.6	0.354	0.738	**0.779**	0.662	0.743	**0.775**
	v3.1 - v3.6	0.905	0.941	**0.972**	**0.949**	0.929	0.939
	v3.4 - v3.6	0.925	0.952	**0.981**	**0.964**	0.941	0.943
Findutils	v4.2.33 - v4.6.0	0.511	0.688	**0.731**	0.631	0.705	**0.746**
	v4.4.1 - v4.6.0	0.736	0.813	**0.912**	**0.898**	0.881	0.887

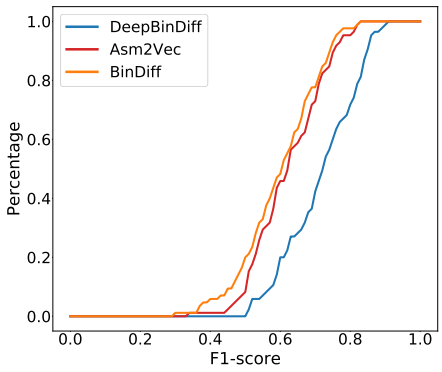


Figure 5.4: v5.93 compared with v8.30

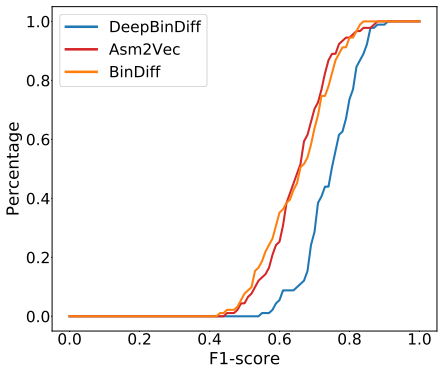


Figure 5.5: v6.4 compared with v8.30

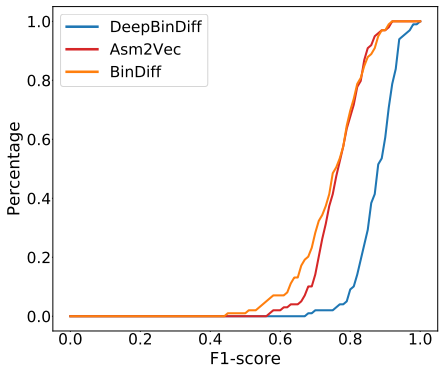


Figure 5.6: v7.6 compared with v8.30

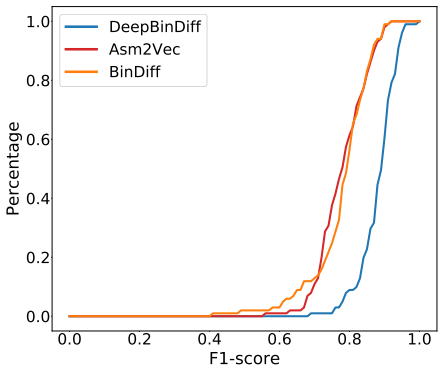


Figure 5.7: v8.1 compared with v8.30

Figure 5.8: Cross-version Diffing F1-score CDF for Coreutils

still enjoys a high precision rate, sometimes higher than DeepBinDiff.

One thing we can see from the results is that cross-optimization level binary diffing is more difficult than cross-version diffing since the recall and precision rates are lower. This is because of the compiler optimization techniques could greatly transform the binaries. However, we anticipate DeepBinDiff and Asm2Vec to achieve better results if trained with more samples.

Table 5.2: Cross-optimization level Binary Diffing Results

		Recall			Precision		
		BinDiff	Asm2Vec	DeepBinDiff	BinDiff	Asm2Vec	DeepBinDiff
Coreutils	v5.93 O1 - O3	0.571	0.556	**0.652**	0.638	0.537	**0.668**
	v5.93 O2 - O3	0.837	0.905	**0.976**	**0.944**	0.855	0.932
	v6.4 O1 - O3	0.576	0.589	**0.676**	0.646	0.569	**0.697**
	v6.4 O2 - O3	0.838	0.899	**0.978**	**0.954**	0.848	0.939
	v7.6 O1 - O3	0.484	0.627	**0.668**	0.674	0.602	**0.704**
	v7.6 O2 - O3	0.840	0.898	**0.953**	**0.944**	0.845	0.913
	v8.1 O1 - O3	0.480	0.628	**0.673**	0.677	0.601	**0.713**
	v8.1 O2 - O3	0.835	0.868	**0.921**	**0.942**	0.839	0.901
	v8.30 O1 - O3	0.508	0.516	**0.607**	0.620	0.495	**0.638**
	v8.30 O2 - O3	0.842	0.884	**0.952**	**0.954**	0.832	0.903
Diffutils	v2.8 O1 - O3	0.467	0.779	**0.831**	0.613	0.755	**0.828**
	v2.8 O2 - O3	0.863	0.955	**0.979**	0.953	0.936	**0.966**
	v3.1 O1 - O3	0.633	0.801	**0.816**	0.655	0.647	**0.775**
	v3.1 O2 - O3	0.898	0.902	**0.943**	**0.966**	0.925	0.964
	v3.4 O1 - O3	0.577	0.712	**0.754**	0.708	0.698	**0.715**
	v3.4 O2 - O3	0.903	0.911	**0.935**	0.953	0.943	**0.967**
	v3.6 O1 - O3	0.735	**0.876**	0.865	0.715	0.811	**0.853**
	v3.6 O2 - O3	0.919	0.954	**0.962**	**0.966**	0.922	0.952
Findutils	v4.233 O1 - O3	0.633	0.695	**0.783**	0.768	0.637	**0.799**
	v4.233 O2 - O3	0.933	0.952	**0.983**	0.968	0.931	**0.981**
	v4.41 O1 - O3	0.677	0.715	**0.821**	0.731	0.677	**0.882**
	v4.41 O2 - O3	0.839	0.912	**0.951**	0.964	0.952	**0.967**
	v4.6 O1 - O3	0.563	0.636	**0.763**	0.633	0.721	**0.791**
	v4.6 O2 - O3	0.958	0.935	**0.961**	0.932	0.915	**0.954**

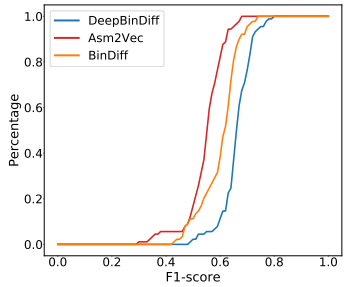


Figure 5.9: v5.93O1 compared with v5.93O3

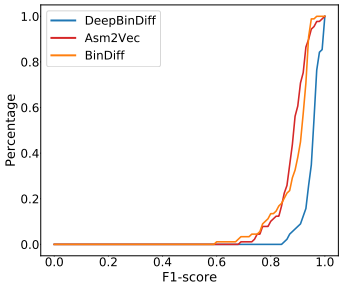


Figure 5.10: v5.93O2 compared with v5.93O3

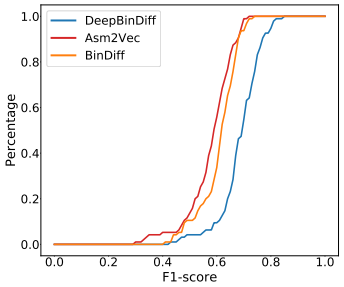


Figure 5.11: v6.4O1 compared with v6.4O3

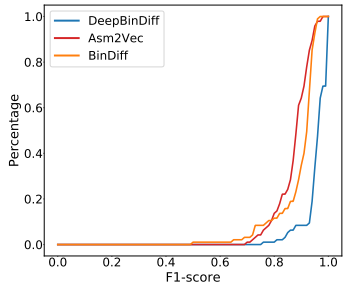


Figure 5.12: v6.4O2 compared with v6.4O3

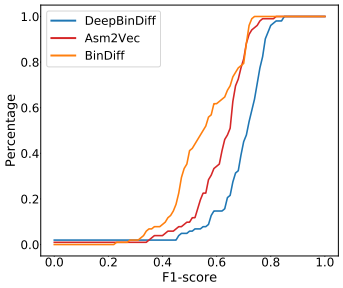


Figure 5.13: v7.6O1 compared with v7.6O3

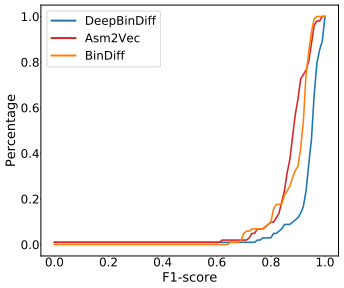


Figure 5.14: v7.6O2 compared with v7.6O3

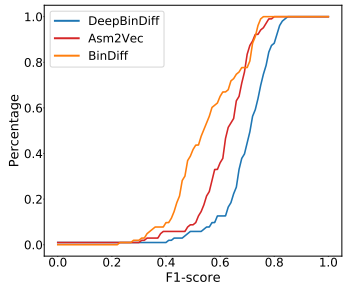


Figure 5.15: v8.1O1 compared with v8.1O3

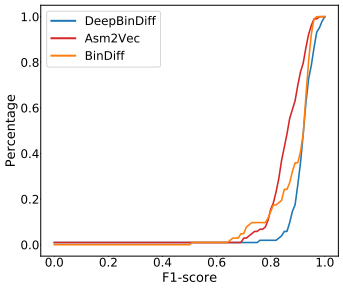


Figure 5.16: v8.1O2 compared with v8.1O3

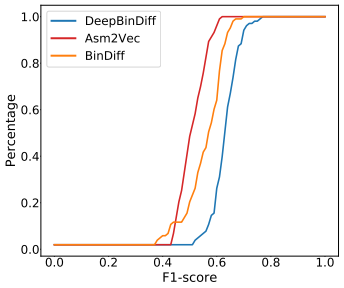


Figure 5.17: v8.30O1 compared with v8.30O3

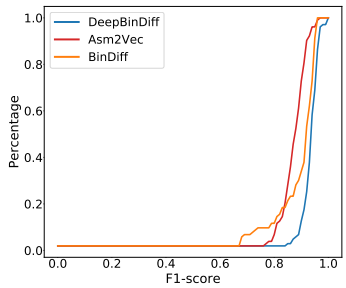


Figure 5.18: v8.30O2 compared with v8.30O3

Figure 5.19: Cross-optimization level Diffing F1-score CDF for Coreutils

5.7.5 Efficiency

We then conduct experiments to show the efficiency of our system. The running time of DeepBinDiff can be split into four parts: training time, preprocessing time, embedding generation time and matching time.

Training Time We train our token embedding generation model with the binaries in our dataset. Random walks are generated for each binary to produce one article for training our model. We stop the training for each binary when loss converges or it hits 10000 steps. In total, It takes about 30 hours for our machine to finish the whole training process. We could retrain our model when new binaries are fed into DeepBinDiff. For comparison, Asm2Vec also needs this training time to generate its model while BinDiff does not need any training time. Note that training the model is only an one-time effort and the training process could be significantly accelerated if GPUs are used.

Preprocessing Time DeepBinDiff relies on IDA pro for ICFG generation. It takes only an average of 12.264s DeepBinDiff to finish the graph generation on one binary. Then, our system applies the pre-trained model to generate token embeddings and calculates the feature vectors for each basic block. Applying the model and calculating the feature vector only takes less than 100ms for one binary. BinDiff which also uses IDA pro for preprocessing takes similar time for its preprocessing.

Embedding Generation The most heavy part of DeepBinDiff is the embedding generation which utilizes TADW to factorize a matrix. On average, it takes 591s on average to finish embedding generation for one binary. One way to accelerate the process is to use

a more efficient algorithms other than the Alternating Least Squares (ALS) algorithm for TADW matrix factorization. For example, CCD++ [137] is demonstrated to be 40 times faster than ALS algorithm.

Matching Time K-hop greedy matching algorithm is efficient in that it limits the search space by searching only within the K-hop neighbors for the two matched blocks. On average, it takes DeepBinDiff 45 seconds to finish the matching. BinDiff, for comparison, takes only 3.4s to finish matching since it uses many heuristics to avoid graph matching.

5.7.6 Case Study

Besides the above experiments, we also evaluate DeepBinDiff with real-world vulnerability analysis to showcase its efficacy in practice. Two representative vulnerabilities in OpenSSL [33] are utilized for an in-depth comparison among our tool and the state-of-the-art commercial diffing tool BinDiff.

DTLS Recursion Flaw The first vulnerability (CVE-2014-0221) happens in OpenSSL v1.0.1g and before, gets fixed in v1.0.1h. It is a Datagram Transport Layer Security (DTLS) recursion flaw vulnerability which allows attackers to send an invalid DTLS handshake to OpenSSL client to cause recursion and eventually crash.

Listing 5.1 shows the vulnerability along with the patched code. As listed, patching is made to avoid the recursive call by changing it to a *goto* statement (Ln. 10-11).

Listing 5.1: DTLS Recursion Flaw

```
1  static long dtls1_get_message_fragment() {
2     int i, al;
3     ...
4
5    + redo: ;
6     if((frag_len = fragment(s, max, ok)) {
7        ....
8        if (s->msg_callback) {
9           s->msg_callback(0, s->version)
10          - return dtls1_get_message_fragment(); ;
11          + goto redo;
12       }
```

To analyze this vulnerability, we feed a vulnerable version as well as a patched version of OpenSSL into the diffing tools and see if the tools can generate correct matching between the blocks that contain the vulnerability and the patch.

Partial results from BinDiff and DeepBinDiff are shown in Figure 5.7.6. This matching is hard because in v1.0.1h, the patched function $dts1\_get\_message\_fragment()$ is inlined into another function named $dts1\_get\_message()$. BinDiff matches the vulnerable function in v1.0.1h with its caller in v1.0.1g, leaving the original $dts1\_get\_message()$ in v1.0.1g unmatched.

Figure 5.20 shows the matching blocks from BinDiff. Within the function, BinDiff fails to match the block containing a recursive function call to the block containing a *goto* statement. It mistakenly matched the block to the one which has similar opcodes but with completely different context. Meanwhile, DeepBinDiff finds the correct matching shown in

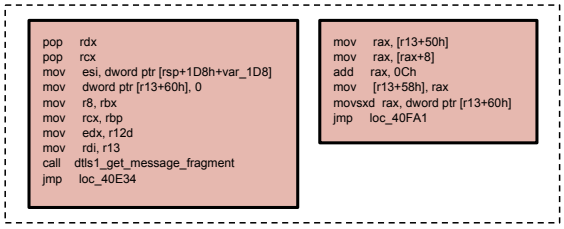


Figure 5.20: BinDiff for Recursion Flaw.

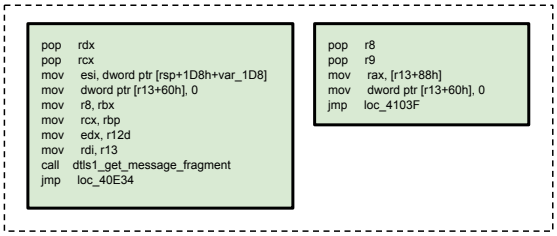


Figure 5.21: DeepBinDiff for Recursion Flaw

Figure 5.21 by considering both the semantics and the program-wide structural information.

Memory Boundary Checking Failure The second vulnerability (CVE-2016-6308) exists in OpenSSL v1.1.0 and before, and gets fixed in v1.1.0a. The program fails to check the length before memory allocation, allowing attackers to allocate excessive amount of memory. As shown in Listing 5.2, the patch inserts a new condition check on top of the original check.

Listing 5.2: Memory Boundary Checking Failure

```
1  static int dtls1_preprocess_fragment() {
2      size_t frag_off;
3      frag_len = msg_hdr->frag_len;
4      if ((frag_off + frag_len) > len) ||
5          + len > max_handshake_message_len(s)) {
6          SSLerr();
7          return SSL_AD_ILLEGAL_PARAMETER;
8      }
9      // memory allocation using len
10     ...
```

For vulnerability analysis, we expect to use binary diffing tool to compare vulnerable binary with the patched one, and identify the patch as a new insertion.

Depicted in Figure 5.22, BinDiff mismatches the vulnerable block with the new condition check block, rendering the real matching block unmatched (shown as white block).

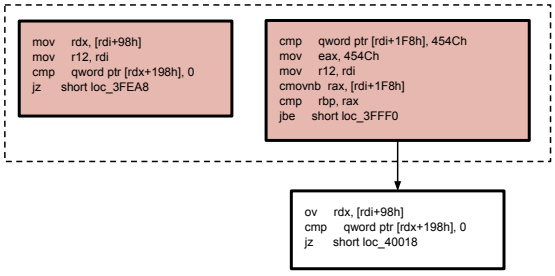


Figure 5.22: BinDiff for Memory Checking

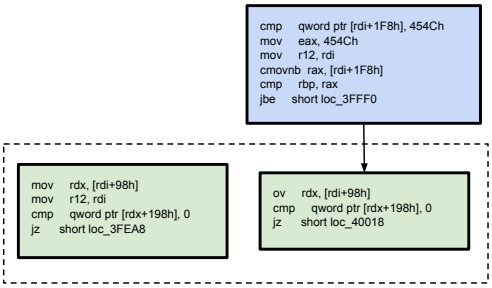


Figure 5.23: DeepBinDiff for Memory Checking

For DeepBinDiff, it successfully matches the blocks and identify the new condition check as a new insertion block.

Chapter 6

Conclusions

In a nutshell, understanding the code changes in programs could help us comprehend the status-quo of Android packing techniques among malware, perform automatic updates to fix existing vulnerabilities for Third-party libraries in Android apps and can improve binary differential analysis for various essential security analyses.

DroidUnpack, a whole-system emulation based Android unpacker that can precisely recover hidden code, is developed to facilitate our study on 6 major commercial packers, 13,566 packed malware samples out of 93,910 Android malware and 3 existing state-of-the-art unpackers in order to better understand the security issues.

LibBandAid solves the outdatedness problem for TPLs in Android apps by automatically generating non-intrusive updates. It effectively patches the security vulnerabilities within libraries with an average of 80.6% success rate and an even higher 94.07% when combined with potentially patchable vulnerabilities.

A novel program-wide code representation learning technique called DeepBinDiff is designed to perform binary diffing in an unsupervised learning fashion. It leverages NLP

techniques to generate token embeddings which are further aggregated to generate block level feature vectors containing semantic information for blocks. It also leverages a K-hop greedy matching algorithm to find optimal matching for the blocks based on the similarity of embeddings.

6.1 Discussions

6.1.1 Final thoughts on Android packing techniques

DroidUnpack is by no means the end of the game but merely a start for future endeavors as the war between packing and unpacking on Android continues. The real problem lies within the design choice of Android system. Unlike iOS which enforces code signing [16] to prohibit app from modification since it was last signed, Android allows the code to be modified even after installation. This feature opens a broad surface for Android packers to perform all kinds of packing techniques without any constraint. Granted, packers are also utilized extensively in legitimate ways for the purpose of protecting intellectual property. However, from the study we surely see packing techniques are currently abused by malware authors, exposing great threats to end users. This situation deserves more thinking for the whole community from a design point of view.

6.1.2 Soundness of LibBandAid

The soundness of our approach results from that of diffing analysis, patch generation and patching respectively. For diffing analysis, we leverage Tracelet Execution [57] technique that demonstrates a 0.99 accuracy in its evaluation to compare two given func-

tions, and extract code changes at statement level. In our case, false positive (statements that are not code changes to be considered as changes) is impossible since we match the exact strings to confirm. Theorically, false negatives are possible, however, we argue that false negative can only cause the potential reduction of success rate but will not bring any correctness or incompatibility issue.

For update generation, the soundness of our impact analysis inherits from the soundness of traditional slicing. The basic scheme strictly follows the definition of *impact* in Section 4.2. However, due to the two optimizations, the *Value-sensitive Differential Slicing* is still sound with respect to the definition of *impact* but may contain over-conservativeness for performance gain.

Based on the soundness analysis of our slicing algorithm, the correctness of updating is ensured by virtue of two reasons. First, LibBandAid introduces absolutely no code changes other than the ones from the new library. We assume the library developers have tested their code before commit. Second, the completeness of each generated update is guaranteed by our slicing algorithm.

6.1.3 Limitations of LibBandAid

There exist several limitations for LibBandAid. To begin with, LibBandAid can only handle Java libraries and Java code changes within these libraries. Therefore, it cannot update native libraries in Android apps. Moreover, non-code changes within Java libraries could bring issues. For example, a version number is recorded in plain text and used to communicate with server as part of the protocol. In this case, the updating from LibBandAid may in fact change the protocol and bring compatibility issues. To solve this

problem, our update generation component has to consider access to the same file as a kind of data dependency. We leave this as a future work.

Second, our slicing algorithm relies on an accurate data dependency analysis that in turn depends on a complete modeling of Java and Android APIs. We manually write models for more than 500 most popular APIs but still can be incomplete. This incompleteness may thwart the soundness of our analysis.

Third, we handle the diffing analysis as a code matching problem and leverage existing research [57] to perform analysis. We argue that this problem is orthogonal to our major focus for updating the TPLs in Android apps. We can definitely make use of the advance in code matching techniques to improve the performance of LibBandAid.

Fourth, although LibBandAid analyzes the library API to collect new exception information, the analysis results in theory can be incomplete. For example, a code change in a TPL's API can call other function outside the library which eventually rises an exception. In this case, we may miss it, jeopardizing the *non-intrusiveness*

Bibliography

[1] Android Statistics. https://www.blog.google/products/android/2bn-milestone/.

[2] Mobile Threat Report Q1 2018. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2018.pdf/.

[3] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[4] apkprotect. https://sourceforge.net/projects/apkprotect/, 2013.

[5] McAfee Labs Threats report Fourth Quarter 2013. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2013.pdf, 2013.

[6] Android developers blog. https://android-developers.googleblog.com/2016/05/whats-new-in-google-play-at-io-2016.html, 2016.

[7] ValerySoftware McAfee. https://securingtomorrow.mcafee.com/mcafee-labs/obfuscated-malware-discovered-google-play/, 2016.

[8] alibaba. http://jaq.alibaba.com/, 2017.

[9] Baidu. http://app.baidu.com/, 2017.

[10] Bangcle. https://dev.bangcle.com/, 2017.

[11] Charger Malware. http://blog.checkpoint.com/2017/01/24/charger-malware/, 2017.

[12] ijiami. http://www.ijiami.cn/, 2017.

[13] Malware repository. https://github.com/ashishb/android-malware, 2017.

[14] Qihoo. http://jiagu.360.cn/, 2017.

[15] Tencent. http://legu.qcloud.com/, 2017.

[16] IOS code signing. https://developer.apple.com/support/code-signing/, 2017.

[17] VIRUSTOTAL. https://www.virustotal.com/, 2017.

[18] Butterknife: Bind Android views and callbacks to fields and methods. https://github.com/JakeWharton/butterknife, 2018.

[19] Dropbox: A Java library for the Dropbox Core API. https://github.com/dropbox/dropbox-sdk-java/, 2018.

[20] EventBus. https://github.com/greenrobot/EventBus, 2018.

[21] F-Droid - Free and Open Source Android App Repository. https://f-droid.org/en/, 2018.

[22] Glide: An image loading and caching library. https://github.com/bumptech/glide, 2018.

[23] Gson: Java serialization library. https://github.com/google/gson, 2018.

[24] Leakcanary: A memory leak detection library for Android and Java. https://github.com/square/leakcanary, 2018.

[25] Okhttp: An HTTP+HTTP/2 client for Android and Java applications. https://github.com/square/okhttp, 2018.

[26] Picasso: A powerful image downloading and caching library for Android. https://github.com/square/picasso, 2018.

[27] Retrofit: Type-safe HTTP client. https://github.com/square/retrofit, 2018.

[28] diffing-with-kam1n0. https://www.whitehatters.academy/diffing-with-kam1n0/, 2019.

[29] GNU Coretuils. https://www.gnu.org/software/coreutils/, 2019.

[30] GNU Difftuils. https://www.gnu.org/software/diffutils/, 2019.

[31] GNU Findutils. https://www.gnu.org/software/findutils/, 2019.

[32] IDA Disassembler and debugger. https://www.hex-rays.com/products/ida/, 2019.

[33] OpenSSL. https://www.openssl.org/, 2019.

[34] zynamics BinDiff. https://www.zynamics.com/bindiff.html, 2019.

[35] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm'13)*, September 2013.

[36] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48. ACM, 2013.

[37] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th international conference on Software engineering*, pages 432–441. ACM, 2005.

[38] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, June 2014.

[39] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.

[40] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. 2011.

[41] Md Tanzirul Azim, Iulian Neamtiu, and Lisa M Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 623–628. ACM, 2014.

[42] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.

[43] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.

[44] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *International Conference on Compiler Construction*, pages 250–254. Springer, 2005.

[45] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):8, 2007.

[46] Shawn Anthony Bohner. A graph traceability approach for software change impact analysis. 1996.

[47] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2013.

[48] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 891–900. ACM, 2015.

[49] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. In *AAAI*, pages 1145–1152, 2016.

[50] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[51] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[52] Mike Chi. *LibDetector: Version Identification of Libraries in Android Applications*. Rochester Institute of Technology, 2016.

[53] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*, pages 2702–2711, 2016.

[54] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th international conference on Information security*, Berlin, Heidelberg, 2011.

[55] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *ACM SIGPLAN Notices*, 51(6):266–280, 2016.

[56] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *ACM SIGPLAN Notices*, volume 52, pages 79–94. ACM, 2017.

[57] Yaniv David and Eran Yahav. Tracelet-based code search in executables. *ACM SIGPLAN Notices*, 49(6):349–360, 2014.

[58] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200. ACM, 2017.

[59] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2019.

[60] Thomas Dullein and Rolf Rolles. Graph-based comparison of executable objects. In *Proceedings of the Symposium sur la Securite des Technologies de L'information et des communications*, 2005.

[61] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security)*. USENIX Association, 2014.

[62] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, October 2010.

[63] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[64] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th Usenix Security Symposium*, August 2011.

[65] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

[66] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *Software Security and Reliability (SERE), 2014 Eighth International Conference on*, pages 78–87. IEEE, 2014.

[67] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[68] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

[69] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[70] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*, pages 238–255. Springer, 2008.

[71] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[72] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium*, 2012.

[73] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services (MobiSys'12)*, June 2012.

[74] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[75] Mark Heimann, Haoming Shen, Tara Safavi, and Danai Koutra. Regal: Representation learning-based graph alignment. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 117–126. ACM, 2018.

[76] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of CCS*, 2011.

[77] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[78] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1037–1049. ACM, 2017.

[79] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, WORM '07, 2007.

[80] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[81] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[82] Jaakko Korpi and Jussi Koskinen. Supporting impact analysis by program dependence graph based forward slicing. In *Advances and innovations in systems, computing sciences and software engineering*, pages 197–202. Springer, 2007.

[83] Harold W Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics (NRL)*, 2(1-2):83–97, 1955.

[84] Nathaniel Lageman, Eric D Kilmer, Robert J Walls, and Patrick D McDaniel. Bindnn: Resilient function matching using deep learning. In *International Conference on Security and Privacy in Communication Systems*, pages 517–537. Springer, 2016.

[85] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society, 2003.

[86] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196, 2014.

[87] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 403–414. IEEE, 2016.

[88] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in android markets. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 335–346. IEEE, 2017.

[89] Yao Li, Weiyang Xu, Yong Tang, Xianya Mi, and Baosheng Wang. Semhunt: Identifying vulnerability type with double validation in binary code. In *SEKE*, pages 491–494, 2017.

[90] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. *IEEE Transactions on Software Engineering*, 44(2):182–201, 2018.

[91] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 2–13. IEEE, 2017.

[92] Jierui Liu, Tianyong Wu, Jun Yan, and Jian 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.

[93] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.

[94] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400. ACM, 2014.

[95] Siqi Ma, David Lo, Teng Li, and Robert H Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 711–722. ACM, 2016.

[96] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 653–656. ACM, 2016.

[97] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23nd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[98] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[99] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *International Conference on Information Security and Cryptology*, pages 92–109. Springer, 2012.

[100] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[101] Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized semantics-based binary diffing with application to malware lineage inference. In *IFIP International Information Security Conference*, pages 416–430. Springer, 2015.

[102] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 259–268. ACM, 2013.

[103] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[104] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 128–137. ACM, 2003.

[105] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114. ACM, 2016.

[106] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. Acm, 2012.

[107] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[108] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 709–724. IEEE, 2015.

[109] Xiaoxia Ren, Barbara G Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings of the 27th international conference on Software engineering*, pages 664–665. ACM, 2005.

[110] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[111] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[112] Raul Santelices and Mary Jean Harrold. Probabilistic slicing for predictive impact analysis. Technical report, Georgia Institute of Technology, 2010.

[113] Raul Santelices, Mary Jean Harrold, and Alessandro Orso. Precisely detecting runtime change interactions for evolving software. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 429–438. IEEE, 2010.

[114] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.

[115] Monirul I. Sharif, Vinod Yegneswaran, Hassen Saïdi, Phillip A. Porras, and Wenke Lee. Eureka: A Framework for Enabling Static Malware Analysis. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2008.

[116] Shashi Shekhar, Michael Dietz, and Dan S Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, volume 2012, 2012.

[117] SophosLabs. Anti-emulation techniques. `https://news.sophos.com/en-us/2017/04/13/android-malware-anti-emulation-techniques/`, 2017.

[118] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. *ACM SIGPLAN Notices*, 42(6):112–122, 2007.

[119] Tim Strazzere. Android hacker protection level 0. `https://www.defcon.org/images/defcon-22/dc-22-presentations/Strazzere-Sawyer/DEFCON-22-Strazzere-and-Sawyer-Android-Hacker-Protection-Level-UPDATED.pdf`, 2014.

[120] Tim Strazzere. android-unpacker. `https://github.com/strazzere/android-unpacker`, 2015.

[121] Mengtao Sun and Gang Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 165–176. ACM, 2014.

[122] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[123] Xabier Ugarte Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *SSP 2015, IEEE Symposium on Security and Privacy, May 18-20, 2015, San Jose, CA, USA*, 2015.

[124] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1225–1234. ACM, 2016.

[125] Haoyu Wang and Yao Guo. Understanding third-party libraries in mobile app analysis. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 515–516. IEEE, 2017.

[126] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 319–330. IEEE Press, 2017.

[127] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 25–36. ACM, 2014.

[128] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS'14)*, Scottsdale, AZ, November 2014.

[129] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[130] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[131] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

[132] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*, pages 462–472. IEEE Press, 2017.

[133] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[134] Cheng Yang, Zhiyuan Liu, Deli Zhao, Maosong Sun, and Edward Y Chang. Network representation learning with rich text information. In *IJCAI*, pages 2111–2117, 2015.

[135] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan,November 2-4, 2015. Proceedings*, 2015.

[136] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, November 2013.

[137] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. Parallel matrix factorization for recommender systems. *Knowledge and Information Systems*, 41(3):793–819, 2014.

[138] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS'14)*, Scottsdale, AZ, November 2014.

[139] Mu Zhang and Heng 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'14)*, San Diego, CA, February 2014.

[140] Mu Zhang and Heng Yin. Efficient, Context-aware Privacy Leakage Confinement for Android Applications Without Firmware Modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS'14)*, 2014.

[141] Mu Zhang and Heng Yin. Efficient, Context-aware Privacy Leakage Confinement for Android Applications Without Firmware Modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS'14)*, 2014.

[142] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 9–18. ACM, 2013.

[143] Xuewen Zhang, Yuanyuan Zhang, Juanru Li, Yikun Hu, Huayi Li, and Dawu 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.

[144] Yiji Zhang and Raul Santelices. Prioritized static slicing for effective fault localization in the absence of runtime information. Technical report, Technical Report TR 2013-06, CSE, U. of Notre Dame, 2013.

[145] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152. IEEE, 2018.

[146] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: Toward extracting hidden code from packed android applications. In *Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, 2015.

[147] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland'12)*, May 2012.

[148] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[149] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, February 2012.

[150] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *NDSS*, 2019.