POLITECNICO DI MILANO
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

SimDroidUI: a new method of UI-based Clustering of Android applications

Necst Lab
Novel, Emerging Computing System
Technologies Laboratory

Relatore: Prof. Stefano Zanero
Correlatore: Prof. Federico Maggi

Tesi di Laurea di:
Giuseppe Palese, matricola 783238

Anno Accademico 2013-2014

To my family, to my friends...

Abstract

With the growing popularity of smartphones, and consequently of application stores, a method for comparing Android applications is needed. In response, researchers proposed various approaches to capture application similarity by comparing source code. Because they rely on calculating differences between code used by the developers, such approaches are good for detecting pirated applications, code reuse, and for bug fixing. However, they are ineffective where a comparison of just the front-end of the application is needed; this is required to compare applications with similar usage and similar User Interface (UI), even with different code structure. A method for analyzing application front-end has been proposed in previous works, but this method often needs to execute applications in a remote emulator.

In light of this, we propose a new approach for similarity analysis among Android applications by comparing the application front-end without executing Android application. The key intuition is that we can consider application UIs as hierarchies of view elements. This assumption allows us to compare application front-ends by comparing sequences of elements. Two applications can be considered as much visually similar as many part these hierarchies they have in common. Moreover, we can improve our approach by grouping together applications with the same UI and by obtaining a summary of the common UI elements. As a consequence, the operation of clustering brings to quickly identifying series of applications similar to a given one without comparing a new sample with the whole database.

We experimentally demonstrate that our approach builds ”clusters” that are good representation of UI similarity. Our results show that *SimDroid-UI* is able to identify application similarity more quickly and effectively than previous works. Finally, we prove that cluster created by *SimDroid-UI* tends to put together applications with the same type of usage for the final user, rather than similarity at source code level.

Sommario

Negli ultimi anni siamo stati testimoni di una notevole diffusione di smartphone e conseguentemente di numerosi negozi virtuali per la distribuzione di applicazioni Android.

L'enorme quantità di applicazioni ormai esistente sui mercati rende necessario il ricorso a metodi sempre più evoluti per il loro confronto e la loro classificazione. Accanto al mercato ufficiale di Google Play sono sorti e si sono progressivamente diffusi numerosi mercati alternativi che mettono a disposizione degli utenti variegate applicazioni non facilmente controllabili. Questo solleva numerosi problemi di sicurezza e di difesa della proprietà intellettuale, poiché non è agevole individuare, tra le stesse, quelle malevole o quelle che violano i diritti d'autore.

In letteratura esistono diversi metodi, basati sul confronto tra applicazioni Android, per fronteggiare tali problematiche. La maggior parte di essi analizza il codice sorgente. Tuttavia, nel tempo tali metodi hanno mostrato limiti e si sono rivelati inefficaci in relazione alla necessità di confrontare le Interfacce Utente. Sui mercati sono infatti rinvenibili applicazioni che, pur apparendo graficamente simili, sono caratterizzate da codici sorgente differenti tra loro.

Attualmente esiste un metodo che compara le interfacce utente delle applicazioni basandosi esclusivamente sull'analisi di immagini e non sulla struttura delle interfacce; esso per di più richiede lunghi tempi di esecuzione.

Il nostro approccio propone un nuovo ed efficiente metodo di misura della similarità tra due applicazioni Android che guarda la struttura dell'interfaccia utente e che risulta scalabile in caso di ampi database di applicazioni. L'idea sottostante a detto approccio è quella che una qualsiasi applicazione Android è rappresentata mediante una serie di layout proposti all'utente. Questi ultimi sono paragonabili a degli alberi tra i quali sussiste un insieme di relazioni del tipo padre-figlio. Applicazioni con la stessa Interfaccia utente presenteranno alberi simili. Ad esempio due applicazioni che presentano entrambe, in una schermata, una lista di campi testuali, saranno rappresentate da simili relazioni padre-figlio, dove i figli sono i campi testuali e il padre è l'oggetto che li raggruppa.

Abbiamo esaminato lo stato dell'arte del tree mining, focalizzandoci sui migliori metodi di confronto tra alberi di elementi. Questi ricercano,

all'interno di un database di alberi (chiamato foresta), le strutture ricorrenti, rappresentate da sotto-alberi. Due alberi, e dunque due applicazioni, risulteranno tanto più simili, quanti più sotto-alberi essi hanno in comune. Essendo la lista di pattern di una applicazione considerabile come insiemi di elementi, è stato possibile ricorrere alla Similarità di Jaccard, ben nota in letteratura, che esprime un valore oggettivo di similarità tra due insiemi.

Più in dettaglio, con *SimDroid-UI* , abbiamo implementato un metodo di raggruppamento (clustering) di applicazioni Android basato sulla similarità di Jaccard.

In sostanza, *SimDroid-UI* offre due funzionalità: (1) raggruppa un database di applicazioni rispetto alla loro similarità visiva e (2) ricerca le applicazioni più simili partendo da un nuovo campione.

A partire dunque dai raggruppamenti creati, *SimDroid-UI* ricerca gli elementi simili rispetto ad una specifica applicazione, permettendo di ritrovare velocemente, all'interno di un ampio database di applicazioni, la lista delle applicazioni più simili al campione.

La novità del nostro metodo sta nel fatto che, a differenza dei metodi di similarità già presenti in letteratura, non ha bisogno di confrontare ripetutamente le singole applicazioni con il nuovo campione, in quanto utilizza una rappresentazione tramite cluster che permette di individuare subito la parte del database contenente le applicazioni più simili.

Abbiamo quindi confrontato *SimDroid-UI* con pHash e Androsim, due dei principali metodi di ricerca di applicazioni simili, utilizzando Puppetdroid, un progetto di analisi dinamica di malware Android. Esso si basa sulla comparazione visiva per simulare l'utilizzo umano di una applicazione Android e risulta tanto più accurato quanto più efficiente è il metodo di similarità utilizzato.

I risultati sperimentali hanno dimostrato che il nostro metodo riesce a produrre gruppi di applicazioni che ben rappresentano la similarità delle singole applicazioni. Inoltre, i raggruppamenti si avvicinano alla divisione semantica che un utente fa rispetto alla tipologia di utilizzo. Abbiamo provato che *SimDroid-UI* è più efficace nella ricerca di applicazioni simili visivamente, confrontandolo con altri approcci esistenti in letteratura. Infine, i test hanno evidenziato come il nostro metodo sia più veloce rispetto a quelli che non utilizzano tecniche di clustering e a quelli che necessitano dell'esecuzione dell'applicazione.

Vero è che non esiste un metodo unico di calcolo dello similarità tra due applicazioni, ma quello che si sceglie di adottare è spesso il più funzionale allo scopo della ricerca. Il nostro metodo può infatti facilmente essere affiancato ai metodi già presenti, per fornire un valore di similarità tra due applicazioni da un diverso punto di vista.

Riassumendo, i contributi originali di questa tesi sono i seguenti:

> abbiamo proposto un nuovo metodo per valutare la similarità tra due

applicazioni Android.

> abbiamo proposto un metodo di raggruppamento di applicazioni basato su similarità visiva;

> abbiamo proposto un metodo di ricerca all'interno di un dataset efficace e scalabile;

> abbiamo implementato il nostro approccio in *SimDroid-UI* , un tool che offre all'utente un servizio di ricerca di applicazioni simili tra loro;

> abbiamo valutato sperimentalmente il sistema, dimostrando che i raggruppamenti creati sono validi e riescono a rappresentare le varie tipologie di applicazioni presenti sul mercato.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In past years we were witness to a large spread of smartphones and, as consequence, of a large number of virtual markets for the distribution of Android applications.

Because of the huge amount of applications available on the markets, the need for methods to compare and categorize applications arose. Besides the Google Play official market, lots of alternative markets grew; applications in these markets are unlikely to be under control. This creates problems of security and intellectual property rights, because it is difficult to find malicious and pirated applications.

In literature, different methods to face such problems in comparing Android applications exist, and they are all based on the comparison between Android applications. Most of them analyze source code.

However, these approaches are ineffective to compare User Interface (UI) more than code. In the market, it is possible to find applications that are similar with respect to the UI, although they have very different source code.

Currently, a method that compares UI exists, focusing on comparing images produced during execution (without dealing with the UI structure); the core problem of this approach is that it requires emulation of the applications, thus comparison is very slow.

Our work proposes a new and efficient way of measuring similarity between two Android applications; it focuses on the structure of the application UI and has the advantage of performance escalation with respect to the application database size.

The idea behind our approach is that an Android application is a combination of layouts that are proposed to the user. These layouts can be represented as a tree structure, composed by series of parent-child relationships. Applications with the same UI will have similar tree structure. For example, considering two applications that present both a list of text field; the corresponding structures will be similar, as they will contain both a main element that groups the type of elements.

We examined the Tree Mining state of the art, focusing on the best methods to compare trees. Most of them search recurrent substructures into a "forest" of trees. Two trees (and so two applications), can be considered to be the more similar the more substructures they have in common. Because the list of patterns that an application includes can be considered as sets, we could use Jaccard Similarity, a well-known similarity measure between two sets. Furthermore, *SimDroid-UI* implements a method for grouping (clustering) applications based on the Jaccard Similarity.

SimDroid-UI offers two functionalities to the final user: (1) groups a database of applications with respect to visual similarity and (2) searches most similar applications to a new sample.

Using clusters created in the first phase, *SimDroid-UI* searches for similar elements to a new specific applications not in the current database, leading to quickly find, inside a large dataset, a list of the most similar applications.

The novelty of our method is that, unlike the methods of similarity already present in the literature, it does not need to repeatedly compare the individual applications with the new sample, as it uses a representation of clusters that allows to identify immediately the part of the database containing the most similar applications.

Then we compared *SimDroid-UI* with pHash and Androsim, two of the main methods for searching similar applications, using Puppetdroid, a dynamic analysis tool that detects Android malwares. Based on visual comparison, it simulates the human use of an Android application and the more efficient is the method of similarity used, the more accurate the tool results.

The experimental results showed that our method is able to produce groups of applications that well represent the similarity of the individual applications. We proved that *SimDroid-UI* is more effective in the search of visually similar applications, comparing it with other existing approaches in the literature. Finally, tests highlighted that our method is faster than those that do not use clustering techniques and those that require the execution of the application. The truth is that there is no unique method for calculating similarity between two applications, but often the most effective method in relation to the research purpose is choosen. Our method can be easily used in conjunction with other methods already present, providing a similarity value from a different perspective.

Summarizing, the innovative contributions of the our thesis are the followings:

> we propose a new approach to evaluate similarity between two Android applications;

> we propose an efficient method for grouping applications considering their visual similarity;

> we propose an effective and scalable searching method within a dataset;

> we implemented our approach in a tool, called *SimDroid-UI* , that offers a similar application searching service to the user;

> we experimentally evaluated the system, demonstrating the validity of the created groups and proving that they are able to represent in some way the types of applications in the markets.

The document is organized as follows.

In Chapter 2 we introduce the Android platform, focusing on security issues and malware analysis. We describe Puppetdroid, a malware analysis tool that uses similarity comparison to perform analysis. Then we analyze the state of the art for our research, focusing on the problem of comparing Android applications.

In Chapter 3 we describe our approach and the problems we faced; we represent application UI as a tree and we compare applications by comparing structures; in order to achieve our goal, we defined a similarity measure and the clustering process.

In Chapter 4 we illustrate the details of our implementation, *SimDroid-UI* , describing how the files where elaborated and transformed and how we grouped applications using a clustering method.

In Chapter 5 we present our test evaluation, performing a comparison between intra and extra cluster similarity; then we compare our approach with two main alternative methods for application similarity, pHash and Androsim.

In Chapter 6 we discuss the whole system implemented, presenting conclusion of our work; finally we describe the future work planning schedule.

Chapter 2

Background and state of the art

This chapter is organized as follows.

In Section 2.1 we introduce Android platform, presenting a general overview of the operating system (section 2.1.2) and of the markets (section 2.1.3). Then we focuses on the issues and open challenges (section 2.1.5).

In Section 2.2 we describe the Puppetdroid Project, a dynamic analysis software that needed a comparison of UI similarity and that we used to testing purpose.

In Section 2.3 we analyze the State of the Art in Android application similarity measurement separating code-based approaches (Section 2.3.1) from non code-based approaches (Section 2.3.2).

2.1 The Android platform

2.1.1 Overview

Due to its characteristics [1, 2, 3, 4], Android is the world's most popular mobile operating system. The increasing interest arises from two core aspects: its open source nature and its architectural model.

Being an open-source project, Android can be fully analyzed and understood, which enables feature comprehension, bug fixing, further improvements regarding new functionalities and, finally, porting to new hardware.

Another aspect that is important to consider is Android *Virtual Machine* (VM) environment. Android applications are Java-based, and this factor entails the use of a VM environment, with both its advantages and known problems.

Android was made publicly available during the fall of 2008. Since its official public release, it has captured the interest of companies, developers and the general audience. The platform has been constantly improved over

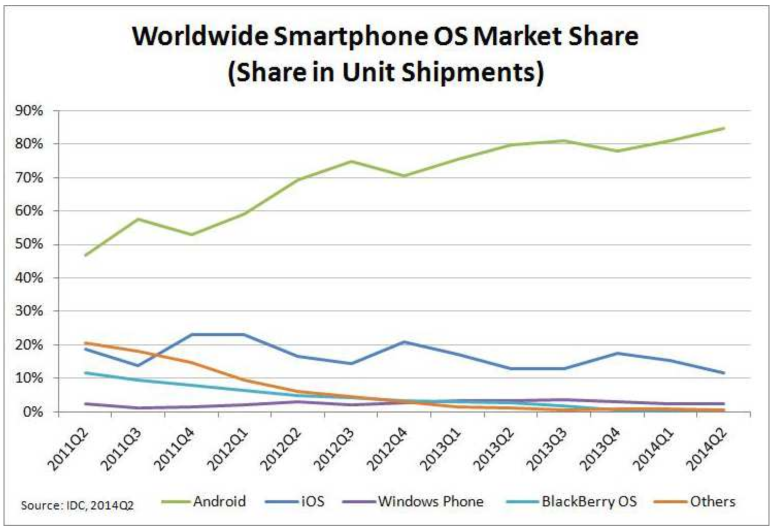


Figure 2.1: Android marketshare from 2nd quarter 2011 to 2nd quarter 2014

time boths in terms of features and supported hardware, even extendeing it to new types of devices different from the originally intended mobile ones.

Android powers more than one billion smartphones and tablets. On September 2014 its market share was 47% [5]. It continues to dominate the global smartphone market, with over 255 million units shipped and nearly 85% of the market share in the second quarter of 2014 [1] [6].

2.1.2 Android Architecture

Android is an open-source software architecture provided by the Open Handset Alliance, a group of 84 technology and mobile companies whose objective is to provide a mobile software platform.

The Android platform includes an operating system, middleware and applications. As for the features, Android incorporates the common features found nowadays in any mobile device platform, such as: application framework reuse, integrated browser, optimised graphics, media support, network technologies, etc. [7]

The Android architecture, depicted in Figure 2.2 [8], is composed by five layers: *Applications*, *Application Framework*, *Libraries*, *Android Runtime* and finally the *Linux kernel.*

The uppermost layer, the Applications layer, provides the core set of applications that are commonly offered out of the box with any mobile device.

The Application Framework layer provides the framework *Application*

[1]Samsung was the largest vendor of Android-based devices, followed by Huawei, Lenovo and LG.

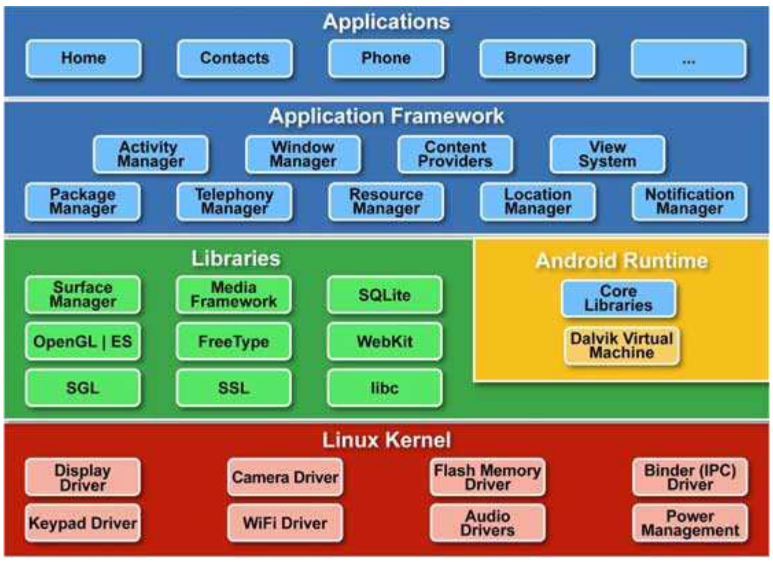


Figure 2.2: The Android architecture layers and modules

Programming Interfaces (APIs) used by the applications running on the uppermost layer. Besides the APIs, there is a set of services that enable access to Android core features such as graphical components, information exchange managers, event managers and activity managers, as examples.

Below the Application Framework layer, there is another layer containing two important parts: Libraries and the Android Runtime. The libraries provide core features to the applications. Among all the libraries provided, the most important are *libc*, the standard C system library, tuned for embedded Linux-based devices; the Media Libraries, which support playback and recording of several audio and video formats; Graphics Engines; Fonts; a lightweight relational database engine and 3D libraries based on OpenGL ES.

Regarding the Android Runtime, besides the internal core libraries, Android provides its own VM, as previously stated, named *Dalvik*. Dalvik was designed from scratch and it is specifically targeted for memory-constrained and CPU-constrained devices. It runs Java applications but unlike the standard Java VMs, which are stack-based, Dalvik is an infinite register-based machine.

Dalvik uses its own byte-code format name *Dalvik Executable* (.dex), with the ability to include multiple classes in a single file. It is also able to perform several optimizations during dex generation when concerning the internal storage of types and constants by using principles such as minimal repetition, per-type pools and implicit labeling.

By applying these principles, it is possible to have dex files smaller than

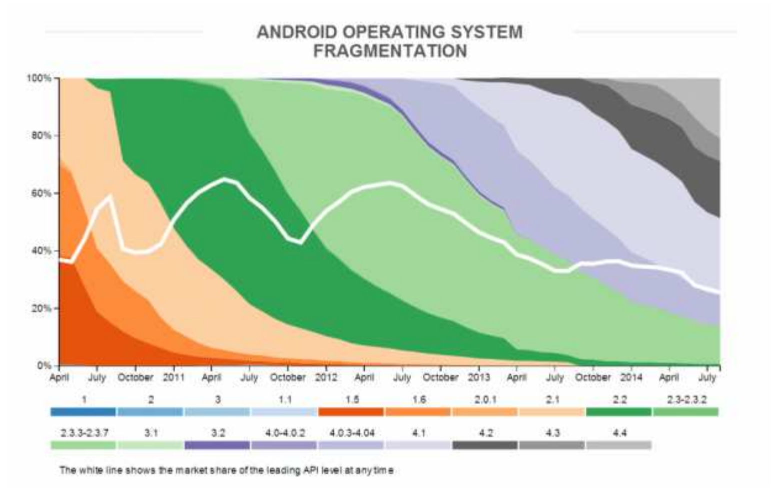


Figure 2.3: Android Operating System fragmentation from April 2012 to July 2014

a typical Java archive (jar) file. During install time, each dex file is verified and optimizations such as byte-swapping and padding, static-linking and method in-lining are performed in order to minimize the runtime evaluations and at the same time to avoid code security violations.

The Linux kernel, version 2.6, is the bottommost layer and is also a hardware abstraction layer that enables the interaction of the upper layers with the hardware layer via device drivers. Furthermore, it also provides the most fundamental system services such as security, memory management, process management and network stack.

Fragmentation

The openness of the Android Platform allows the manufacturers and carriers to alter it at will, making arbitrary customizations to fit the OS to their hardware and distinguish their services from what their competitors offer. Further complicating this situation is the fast pace with which the *Android Open Source Project* (AOSP) upgrades its OS versions.

Figure 2.3 shows the Operating System fragmentation of Android platform from April 2012 to July 2014. Since 2009, 19 official Android versions have been released. Most of them have been heavily customized, which results in tens of thousands of customized Android branches coexisting on billions of mobile phones around the world. This fragmented ecosystem not only makes development and testing of new apps across different phones a challenge, but it also brings in a plethora of security risks when vendors and carriers enrich the system's functionalities without fully understanding the security implications of the changes they make. Figure 2.5 shows the Android Operating System version distribution at September 2014.

Version	Codename	Distribution
2.2	Froyo	0.6 %
2.3.X	Gingerbread	9.8 %
4.0.X	Ice Cream Sandwitch	8.5 %
4.1.X	Jelly Bean	22.8 %
4.2.X	Jelly Bean	20.8 %
4.3.X	Jelly Bean	7.3 %
4.4.X	Kitkat	30.2 %

Figure 2.4: Android Platform Versions distribution

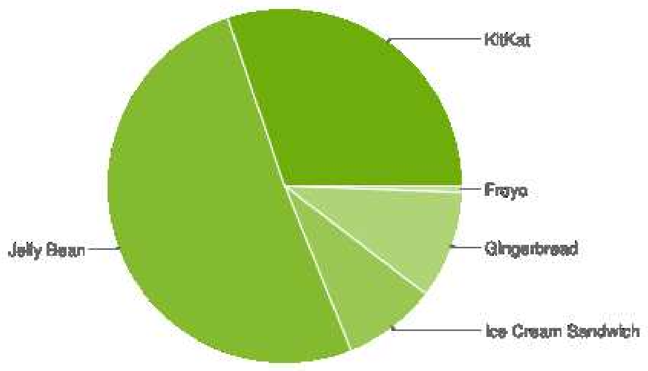


Figure 2.5: Android Platform Versions distribution cake

Because device makers are free to enhance the Android operating system with user interface additions, and because vendors do not update all their customers' devices with the latest version of the Android OS, there is a huge number of Android hardware/software combinations on the market. Not only it add confusion among users, it becomes increasingly difficult for the developer to guarantee the 100% application compatibility on every installed device [9, 10]. Figure 2.6 shows screen resolution fragmentation among device distribution. This bring developer to customize application based to device hardware, software version, screen resolution and screen dimension.

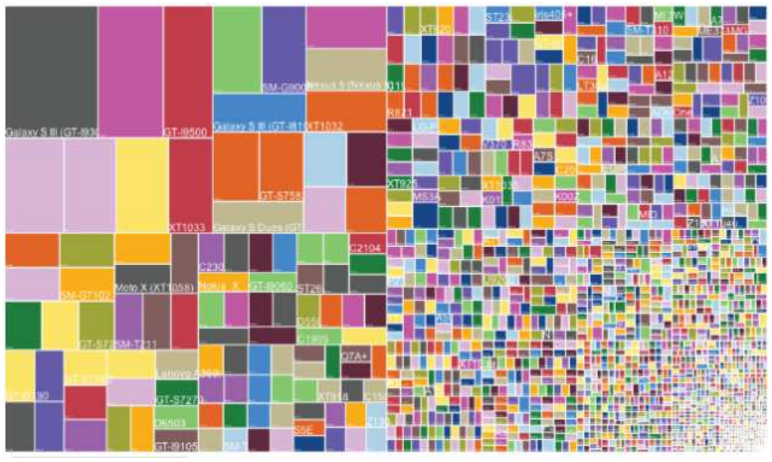


Figure 2.6: Android screen resolution fragmentation

2.1.3 Android markets

Android applications are distributed both through the official store of Google Play and through many so-called *alternative marketplaces*.

Google Play

Google Play, formerly the Android Market, is a digital distribution platform operated by Google. It serves as the official app store for the Android operating system, allowing users to browse and download applications developed with the *Android SDK* and published through Google [11]. Google Play also serves as a digital media store, offering music, magazines, books, movies, and television programs.

Google uses an in-house automated antivirus system, called *Google Bouncer*, to remove malicious applications uploaded on to the marketplace. Bouncer is credited with reducing malware by 40 percent between the first and the second quarter of 2011.

Alternative markets

There are a number of alternative Android marketplaces, web services whose primary purpose is to distribute Android applications. According to this definition, researchers were able to find 89 alternative marketplaces as of June 2013, run by companies or individuals [12].

The *raison d'etre* of such alternative markets are: country gaps (i.e., the Google Play Store is inaccessible from certain countries), promotion (i.e., markets tailored to help users find new interesting applications), and specific needs (i.e., markets that publish applications that would be bounced by the Google Play Store) [12]. But there are also risks connected to the existence of alternative Android app markets, in particular malware. The security policy on different Android app stores will vary. Some will perform safety checks similar to Google, others will not. Other problems you may encounter relate to a poor user experience. There are also app stores that carry pirated versions of apps and games.

2.1.4 Malicious Applications

Lookout mobile security has reported that malware resulted in a loss of US$1 million in 2012. According to a 2014 research study released by RiskIQ, malicious apps introduced through Google Play store have increased 388% between 2011 and 2013. The study also revealed that the number of malicious apps removed annually by Google has dropped drastically, from 60% in 2011 to 23% in 2013. Apps for personalizing Android phones led all categories as most likely to be malicious.

The two types of markets can be analyzed w.r.t. malware and similarity of Android applications, which is an active research topic. This requires analyzing the application package file (APK), a compressed archive that contains resources (e.g., media files, manifest) and code, including Dalvik executables or libraries, or native code (e.g., ARM or x86). Dynamic, static and hybrid program analysis approaches have also been ported to Android [12].

Regarding malware distribution, since the first measurements conducted in 2011 [13], a lot has changed: researchers, security vendors and media continuously raise concerns about the explosive growth of Android malware. According to a recent estimate, as of 2013, companies have invested about US$9 billion in mobile device and network security, and installation of anti-malware software has become a de-facto requirement for mobile devices. Over the years, the number of malware increased significantly and continues to expand.

Since the first detected Android malware in August 2010, well over 300 malware families and more than 650,000 individual samples of malware for Android have been recorded [13], a tiny fraction of the existing types of malware for the traditional PC, but a growing threat. Android malware has grown quickly in a short space of time and looks set to keep growing apace with our use of mobile devices.

Malware authors on traditional platforms use obfuscation techniques like dead code insertion, register reassignment, subroutine reordering, instruction substitution, code transposition, and code integration to evade detection by traditional defenses like antivirus [14], which typically use signature based techniques and are unable to detect the previously unseen malicious executables.

A previous research on malicious applications on popular Android markets showed that in 2011 the majority of malicious or otherwise unwanted Android applications were distributed through alternative marketplaces. The experiments with 204.040 apps collected from five different Android markets in May-June 2011 revealed 211 malicious ones: 32 from the official Android market (0.02% prevalence) and 179 from alternative marketplaces (prevalently ranged from 0.20% to 0.47%). It shows a relatively low malware prevalence on studied Android markets (0.02% on official Android market and 0.20% to 0.47% on other alternative marketplaces). Such prevalence is certainly less than that of malicious web contents. However, due to the centralized role they played in the smartphone app ecosystem, such prevalence, though low, can still compromise a tremendous number of smartphones and cause lots of damages. [14]

Detection methods for attacks on mobile devices have been proposed to reduce the damage from the distribution of malicious apps. However, mechanisms that provide more accurate ways of discerning normal and malicious apps on common mobile devices must be developed [15].

As Android applications become increasingly ubiquitous, we need algorithms and tools to protect applications from *product tampering* and *piracy*, while facilitating valid product updates. Since it is easy to derive Java source code from Android byte code, Android applications are particularly vulnerable to tampering.

To systematically detect malicious apps in existing Android Markets, there are three key goals to achieve: *accuracy, scalability, and efficiency. Accuracy* is a natural requirement to effectively detect malicious apps in current marketplaces with low false positives and negatives; *scalability* and *efficiency* are challenging as we need to accommodate the large number of apps that need to be scanned [16].

The current Android markets rely mainly on two approaches for identifying misbehaving applications: a *review-based* approach and a *reactive approach.* The former generally involves experts manually investigating and reviewing applications for security problems, and the latter leverages user ratings, reportings, and policing to identify problematic applications. Neither of these approaches is scalable and reliable given the hundreds of thousands of applications available. This necessitates ways to quickly and automatically search through large application datasets and reduce the number of possible misbehaving applications to a small set for further examination.

2.1.5 Open challenges

Similarity check

One approach to addressing the problem of identifying misbehaving applications is to determine the similarity among Android applications (by comparing applications from the official Android market to third party markets – *similarity analysis approach*) with the goal of detecting known buggy code patterns and vulnerabilities, repackaged and pirated applications, and known malware in Android markets. Detecting code reuse in Android applications offers a first chance in detecting applications that may negatively impact the user's security and experience or defraud developers of revenue [17].

Unlike Apple's App Store, Android markets tend not to vet applications but rather rely on user feedback. This relaxed policy makes it easier for people to clone, modify, and redistribute applications.

Although considerable research has been conducted on clone detection, unfortunately, existing techniques are not suitable for detecting app clones on Android Markets. Method-level similarity could be used to get similar apps. However, similar apps are not always app clones. Similar apps cause can be one of these three categories [18]:

a) Apps from the same author;

b) Apps developed using the same framework or using common third-party libraries (e.g. advertisement libraries);

c) App clones.

The similarity comparison research has the goal to separate separate c) from a) and b).

Malware analysis

Malware to be analyzed so as to understand the associated risks and intentions. The malicious program and its capabilities can be observed either by examining its code or by executing it in a safe environment.

Static analysis Disassemble/Debugger tools like IDA Pro (`https://hex-rays.com/products/ida/index.shtml`) and Smali (`https://code.google.com/p/smali/`) displays the malware's code as assembly instructions, which provide a lot of insight into what the malware is doing and provide patterns to identify the attackers. Memory dumper tools like ADB (`http://developer.android.com/tools/help/adb`) line used to obtain protected code located in the system's memory and dump it to a file. This is a useful technique to analyze packed executables which are difficult to disassemble.

Binary obfuscation techniques, which transform the malware binaries into self compressed and uniquely structured binary files, are designed to resist reverse engineering and thus make the static analysis very expensive and unreliable. Moreover, when utilizing binary executables (obtained by compiling source code) for static analysis, the information like size of data structures or variables gets lost thereby complicating the malware code analysis [19]. The evolving evasion techniques being used by malware writers to thwart static analysis led to the development of dynamic analysis [20].

Dynamic Analisys Analyzing the behavior of a malicious code (interaction with the system) while it is being executed in a controlled environment (virtual machine, simulator, emulator, sandbox etc.) is called dynamic analysis. Before executing the malware sample, the appropriate monitoring tools like APIMonitor (`http://www.rohitab.com/apimonitor`) or Strace (`http://linux.die.net/man/1/strace`) must be activated.

Various techniques that can be applied to perform dynamic analysis include function call monitoring, function parameter analysis, information flow tracking, instruction traces and autostart extensibility points etc. [19]. Dynamic analysis is more effective as compared to static analysis and does not require the executable to be disassembled. It discloses the malwares' natural behavior which is more resilient to static analysis. However, it is time intensive and resource consuming, thus elevating the scalability issues. The virtual environment in which malwares are executed is different from the real one and the malwares may perform in different ways resulting in artificial behavior rather than the exact one. In addition to this, sometimes the malware behavior is triggered only under certain conditions and can't be detected in virtual environment. Several online automated tools exist for dynamic analysis of malwares, e.g. Droidbox (`https://code.google.com/p/droidbox/`), TaintDroid (`http://appanalysis.org/`), Andrubis (`https://anubis.iseclab.org/`) and DroidScope (`https://code.google.com/p/decaf-platform/wiki/DroidScope`). The analysis reports generated by these tools give in-depth understanding of the malware behavior and valuable insight into the actions performed by them. The analysis system is required to have an appropriate representation for malwares, which are then used for classification either based on similarity measure or feature vectors.

However a large number of new malware samples arriving at anti-virus vendors every day requires an automated approach so as to limit the number of samples that require close human analysis. Several Artificial Intelligence techniques, particularly machine-learning based techniques have been used in the literature for automated malware analysis and classification.

2.2 The PuppetDroid project

Puppetdroid [21] is a project that provides dynamic analysis method by automatic code-exercising and stimulation techniques. The reason behind this project is that a human user is able to exercise certain behaviors of a malware that a dynamic analysis tool by code stimulation fails to unveil.

The first goal of the project is to provide a sandboxed environment to safely perform manual tests on (malicious) applications and, at the same time, record user interaction with the UI of the application. The second goal is to automatically exercise unknown applications, leveraging stimulation traces previously recorded on similar applications.

2.2.1 Approach overview

PuppetDroid's approach is to let applications run on a remote sandbox while users seamlessly interact with their UI as if they were running locally on their devices. More precisely, in order to records stimulation traces, each sandbox uses a remote frame-buffer protocol to collect stimulation traces, which represent the sequence of UI events performed by the user, as well as the list of UI elements actually stimulated during the test. It records raw events from *input* and re-injects them to another input device: such events are correlated to the respective UI elements (e.g., buttons, or other view objects), and information is collected about the behaviors exhibited by the exercised applications, through dynamic analysis.

Furthermore, it leverages the collected stimulation traces to automatically exercise new applications; this operation is due to the fact that using a stimulation close to human behavior, the code tested during dynamic analysis is much greater. The hypothesis is that by re-using stimulation traces better results are obtained (in terms of discovered behaviors) than with random UI testing. A naive approach where is blindly tried to exercise an application with every stimulation trace in the system is not accurate or efficient, but the trace must be executed in application compliant with the concept of UI similarity.

When a new sample is to be analyzed, firstly samples are looked for similar (or equivalent) for which we have a stimulation trace. Then, only stimulation traces of the most similar known application are used.

2.2.2 The basic workflow of PuppetDroid

The user can interact with PuppetDroid in two ways: using the web application (to upload the APKs they want to test, to see the results of tests previously executed tests and to leverage our re-execution functionality) or using the Android application, to manually exercise a given application.

As shown in Figure 2.7 the workflow can be divided into eight steps:

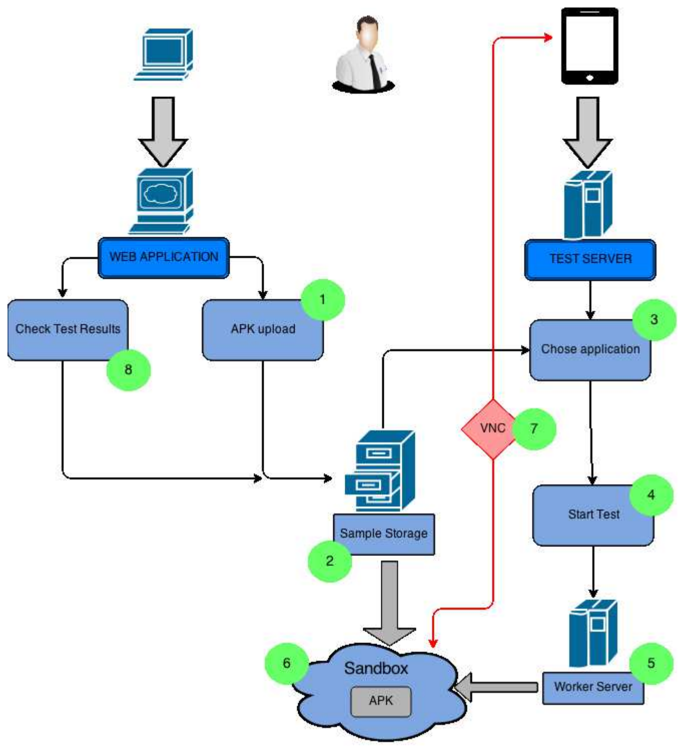


Figure 2.7: Puppetdroid workflow

1. *APK upload*: through the web front-end the user uploads the APK he/she wants to test;
2. *Sample storage*: after performing some consistency and security checks, the uploaded sample is stored into the PuppetDroid sample repository.
3. *Chose application*: through the Android application the user selects the app they wants to test
4. *Start test*: using the chosen application, a new test session can be started. The Android application contacts our main server and sends the request.
5. *Send task to worker*: the main server checks if there is an available worker and, if found, sends the task request to it.
6. *Sandbox initialization*: the selected worker retrieves the APK to be tested from the sample repository and initializes the sandbox that will hold the test session.
7. *VNC session*: when the sandbox is ready, a VNC channel is established with the Android application and the user can interact with the sandbox.

8. *Check test results*: the user can now view test results using the web front-end.

2.2.3 System Details

Recording of stimulation traces The low-level input events generated while the user interacts with an application on their device are recorded. The input events are translated in a sequence of *PointerEvent* or *KeyEvent* messages that are sent to a VNC server. For each event, the *timestamp*, *event type*, *action* (0 is "up", 1 is "down", 2 is "move"), *x position*, *y position* coordinates on the screen, and pressed *button key* are saved.

During recording it is kept track of which view object (e.g., button identifier) consumed each input event during recording, in order to find that same view object during re-execution. For this, it is relied on the *ViewServer*, which allows to "walk" the hierarchy of displayed objects. More precisely, the VNC server performs the following steps when a new input event is received:

1. Process *PointerEvent* message.

2. Send a command to the *ViewServer*, to get the name and hash code of the focused window (i.e., Activity).

3. Retrieve view hierarchy of the window sending DUMPQ command to ViewServer.

4. Search view hierarchy for the deepest-rightmost view object containing the coordinates of the input event.

5. Store the paths to the previously found view nodes.

Two similar applications may have some subtle UI differences that can make a re-execution test fail (e.g., slightly shifted buttons). The collected information are combined to extract the list of paths to the views that actually consumed the touch events. The activity that has consumed each event is logged, and the path to all the deepest nodes in the hierarchy that can consume the touch event; then a sequence of view objects stimulated is built.

Re-execution of stimulation traces For each event in the recorded sequence, the view object and ratio information to properly re-scale the horizontal and vertical coordinates is used. Then, the resulting events are written into the /dev/input device.

Touch events are treated with special care to avoid the following rare corner case, which can occur if the view that receives the input event is not the view that eventually consumes it. More precisely, when a touch event is

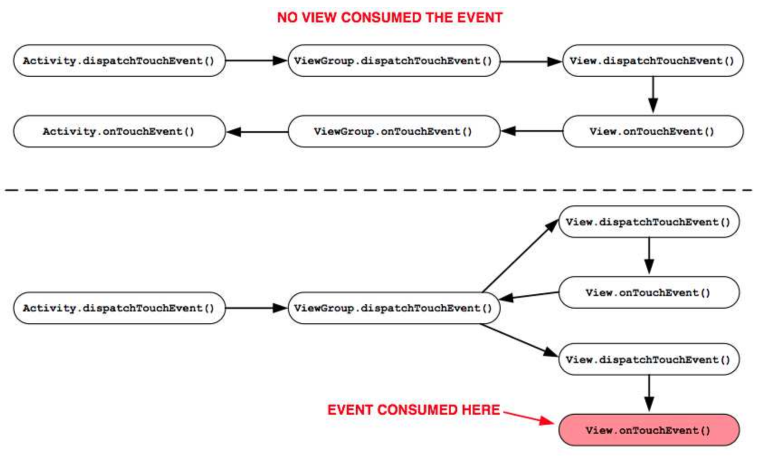

NO VIEW CONSUMED THE EVENT
Activity.dispatchTouchEvent()
ViewGroup.dispatchTouchEvent()
View.dispatchTouchEvent()
Activity.onTouchEvent()
ViewGroup.onTouchEvent()
View.onTouchEvent()
Activity.dispatchTouchEvent()
ViewGroup.dispatchTouchEvent()
View.dispatchTouchEvent()
View.onTouchEvent()
View.dispatchTouchEvent()
EVENT CONSUMED HERE
View.onTouchEvent()

Figure 2.8: Examples of Puppetdroid touch event management

handled by the Android Touch System, the *Activity.dispatchTouch-Event()* method of the currently running Activity is called. This method dispatches the event to the root view in the hierarchy and waits for the result: If no view consumes the event, the Activity calls *onTouchEvent()* in order to consume itself the event before terminating. When a view object receives a touch event, the *View.dispatchTouchEvent()* is called: This method first tries to find an attached listener to consume the event, then tries to consume the event itself calling.

If neither there is a listener nor the *onTouchEvent()* method is implemented, the event is not consumed and it flows back to the parent. When a *ViewGroup* (that contains a series of subviews) receives a touch event, it iterates on its children views in reverse order and, if the touch event is inside the view, it dispatches the event to the child. If the event is not consumed by the child, it continues to iterate on its children until a view consumes the event. If the event is not consumed by any of its children, the *ViewGroup* acts as a View and tries to consume itself the event. Eventually, if it is not able to consume the event it sends back to the parent.

Figure 2.8 shows two examples of touch events management: in the former, the event flows down through the hierarchy, and since it is not consumed by any view, it goes back to the Activity. In the latter, the event is consumed by the second View child of the *ViewGroup* object. The system avoids this corner case because it recorded which activity has consumed each event, and the path to all the deepest nodes in the hierarchy that can consume the touch event.

2.2.4 Open challenge: finding similar applications

In case a stimulation trace for an application A is not available, besides searching by MD5, it is possible to rely on visual similarity to find similar applications. For Puppetdroid, the requirement is to find an application B, for which a stimulation trace exists. The application B must be compliant with the structural implementation of the UI, in terms of pattern to the object (node) that consumed an event. It leverages the concept of visual similarity, that as been implemented in Puppetdroid through screenshot comparison by perceptual hashing computation (Section 2.3.2).

During the execution of both A and B, using the same stimulation trace, the same view objects must be found, in order to stimulate application B emulating a human usage. The presence of a view object in the original sample layout that is not present in the layouts of similar applications is a corner case that can make re-execution incomplete or even fail; it happens because Puppetdroid cannot correctly retrieve the same objects during the process of consuming views. The problem of this implementation is that may occur that two applications produce the same appearance for image comparison but the layouts contain different kind of object.

A more specific but explanatory example is the case of context change; for example, during the stimulation of the original sample, the user clicked on a link embedded in a *TextView* object. Re-executing the test on a similar application, the content of the *TextView* changed, with consequent vanishing of the link. Hence, clicking on the *TextView* in the original sample led to open a new window, while the same click on the similar application did not generate any transition, making the re-execution test fail.

The cause of this limitation is that Phash, by comparing screenshots, does not compare a hierarchy of views, but makes a "blind" search, getting away from the meaning of comparing UI structure. To avoid this specific cases of dynamic layouts, a similarity criterion that can recognize whether two layouts are very similar, yet with a significant tiny variation (e.g, absence of a single, small button) is needed. A further improvement that can avoid re-execution to fails is a method that retrieve applications that have similar flows during execution.

2.3 State of the Art

In this section we will analyze the State of the Art of methods that compare applications and evaluate application similarity. We can distinguish between (1) code-based tool (Section 2.3.1) , that use to compare code produced by the developer to compute similarity and (2) non code-based tool (Section 2.3.2).

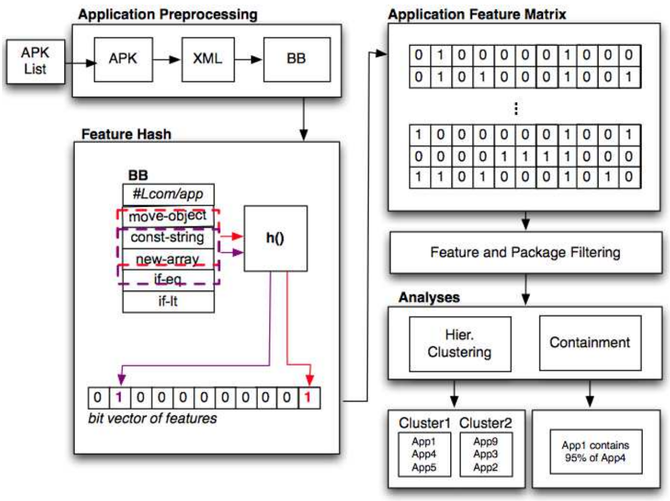


Figure 2.9: The Juxtapp workflow

2.3.1 Code-Based applications

Juxtapp

Juxtapp [22] is an architecture that automatically examines code containment in Android applications.*Code containment* it's defined to be a measure of the relative amount of code in common between two Android applications. The main technique used is feature hashing. Feature hashing is a popular and powerful technique for reducing the dimensionality of the data being analyzed. Using a single hash function, feature hashing compresses the original large data space into a smaller, randomized feature space, in which feature hashing, representation, and pairwise comparison are all efficient.

As shown in figure 2.9, Juxtapp consists of the following steps for analyzing Android applications:

1. application preprocessing;
2. feature extraction;
3. similarity and containment analyses.

For each application APK, its DEX file is extracted and converted into a complete XML representation of the Dalvik program, including program structure. From the XML file, each basic block is extracted and labeled according to which package it came from within the application. For each basic block, only the op-codes are retained and most operands are discarded. The result is a *basic block (BB)* file for each application.

Feature Extraction. *K-grams* of opcodes and feature hashing are used to extract features from applications. From each BB file, k-grams are extracted using a moving window of size k and they are hashed. K-grams across basic blocks are ignored. For each hashed value, the corresponding bits are set in a *bitvector*, which represents the features in the application, to indicate the existence of the k-gram.

Similarity. Two applications are considered to be similar if their bitvector representations of k-grams are similar. The *Jaccard similarity* is calculated dividing the number of features in common between applications by the total number of distinct features the applications have. More formally, the Jaccard similarity between bitvectors A and B is defined as $\mathcal{J}(A, B) = |A \cap B|/|A \cup B|$, and this value represents the similarity between applications. Using this similarity metric, it's possible to perform *agglomerative hierarchical clustering* [23] on all of the bitvectors to group similar applications together into clusters.

Containment. Containment analysis on the bitvectors is performed to determine the percentage of code in common among applications. *Containment* is defined as the percentage of features in application A that exist within application B . This value is computed by dividing the number of features common in both applications by the number of bits in A . More formally, containment $C(A|B) = |A \cap B|/|A|$.

Androsim

Androsim (`http://code.google.com/p/androguard`) leverage *Normalized Compressed Distance* (NCD) to approximate *Kolmogorov complexity* and to calculate the distance between two elements using real world compressors. In particular, given a compressor C, the NCD of two elements A and B is defined by:

$$d_{NCD}(A, B) = \frac{L_A|B - min(L_A, L_B)}{max(L_A, L_B)} \tag{2.1}$$

where L is the length of a string, LA = L(C(A)) and A|B is the concatenation of A and B. The compressor C must be normal, i.e it has to satisfy the 4 inequalities:

1. Idempotency: $C(xx) = C(x), andC(\epsilon) = 0$, where ϵ is the empty string;

2. Monotonicity: $C(xy) \geq C(x)$;

3. Symmetry: $C(xy) = C(yx)$;

4. Distributivity: $C(xy) + C(z) \leq C(xz) + C(yz)$.

Moreover, the compressor must be able to calculate C(x) within an acceptable amount of time. The algorithm, used to calculate the similarity between two applications, works as follows:

> extract the lists of methods from the bytecode of the two samples;

> identify identical methods using an hashing comparison;

> generate signatures for remaining methods;

> identify similar methods using NCD.

The global idea is to associate each method of the first application with others of the second application, excluding identical methods, by using *NCD* with an appropriate compressor. Finally, according to the number of identical methods and the distance between the remaining ones, a similarity score is given as output. The difference computation is based on “strings" and "hashes" properties of elements. In case of Android applications, elements are methods or classes. The "string" is the signature of a method and the "hash" is the sequence of instructions of it.

Androsim uses *control flow graph* (CFG) of the methods along with specific instructions of the CFG such as "if" or "goto". All the instructions, like sparse/packed switch, are translated to "goto" instructions without details. It's interesting to see that even if the basic blocks are in a different order, the Kolmogorov complexity is preserved. If each basic block is reorganized in the signature, it's possible to see that the results are quite the same (so basically the NCD bypasses a basic CFG obfuscation). This tool detects and reports:

> the identical methods;

> the similar methods;

> the deleted methods;

> the new methods;

> the skipped methods.

Moreover, a similarity score (between 0.0 to 100.0) is calculated upon the values of the identical methods (1.0) and the similar methods using the BZ2 compressor. The algorithm just illustrated is very efficient in calculating the similarity between two applications: however pair-wise comparison does not scale if you have to calculate similarity on a large amount of applications.

2.3.2 Other approaches

DStruct

DStruct [24] is a simpler system that examines applications at a higher level than Juxtapp and represents applications by their directory structures. This avoids all of the problems that are associated with the application code and provides another metric for determining application similarity. All of the files and directories and their relationships among each other make up the directory structure of the application. The approach relies on the observation that applications with different functionality typically have different directory structures, and applications that are similar, such as different versions of the same app, will have similar directory structures.

There are several methods to represent the directory structures of applications so that we can apply comparison techniques to them. The leaves of the tree are the files and the non-leaf nodes are the directories of the application. A file leaf is a child of a directory node if the directory contains that file. The method for labeling nodes is crucial as it affects how closely the tree represents the directory structure of the application and, hence, influences the similarity computations among pairs of applications. It explore several methods for labeling nodes:

> using the names of the files;

> extracting the extensions of them;

> computing their *md5* sums;

> using the Linux file command.

Each of these approaches can encounter different cases that lead to false positives and false negatives. Once two applications are represented as trees, we can compute the tree edit distance between them. The *distance* between two trees is considered to be the number of edit operations to transform one tree to another. Edit operations include (1) changing one node label to another, (2) deleting a node, after which all children of the deleted node become children of that node's parent, (3) and inserting a node, after which a consecutive subsequence of siblings among the children of the node's parent become the children of the node. The greater the edit distance, the more different the two trees are from one another. For this purpose, it is possible to use *Zhang-Shasha Algorithm.* This algorithm is used because it is simple and works fast, but there are other algorithms that may perform more accurately or even faster. Then, the percent difference between two trees is calculated as their edit distance divided by the number of nodes of the bigger tree.

$$diff(T1,T2) = \frac{(treeeditdistance)}{max(|T1|,|T2|) * 100\%} \tag{2.2}$$

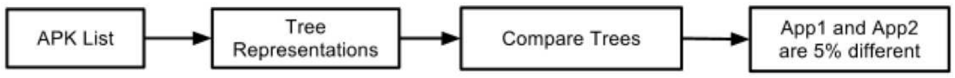


Figure 2.10: The DStruct workflow

It is possible also to use the difference values to perform hierarchical clustering to group similar applications together. In 2.10 the workflow of DStruct is shown.

PHash

Puppetdroid Project relies on an similarity measure of applications that focuses more on *visual similarity* with respect to *coding similarity.* Using Perceptual Hash (`https://www.phash.org`), the system get an hash of app screenshots so that two applications will be as much similar as more the hash computed will be similar.

In practice, to lookup a suitable stimulation trace for application A, the perceptual hash of its screenshots is calculated. Then, in order to find an application B, the system seek for an application which screenshots minimize the Hamming distance from A's screenshots according to their respective perceptual hash. The hashes of the known applications offline are pre-computed, and they are indexed in a tree, which allows lookup in logarithmic time. The system instantiates an emulator, installs the APK of A and leverages the screencap utility to take a screenshot once the application has started, unless it can retrieve it from Google Play.

Given an image in input, a perceptual hashing algorithm creates a metric fingerprint that is robust to image re-scaling, rotation, deformation, skew and compression. Thus, if two images are visually similar, their respective hashes, which are 64-bits unsigned integers, are very close. In particular, perceptually similar images have a hamming distance within bounds that

A *perceptual hash* is a fingerprint of a multimedia file derived from various features from its content. Perceptual image hash functions produce hash values based on the image's visual appearance. A perceptual hash can also be referred to as e.g. a robust hash or a fingerprint. Finally, using an adequate distance or similarity function to compare two perceptual hash values, it can be decided whether two images are perceptually different or not. Perceptual image hash functions can be used e.g. for the identification or integrity verification of images.

The most frequently used functions are:

> Discrete Cosine transform (DCT) based

> Marr-Hildreth operator based;

> Radial variance based;

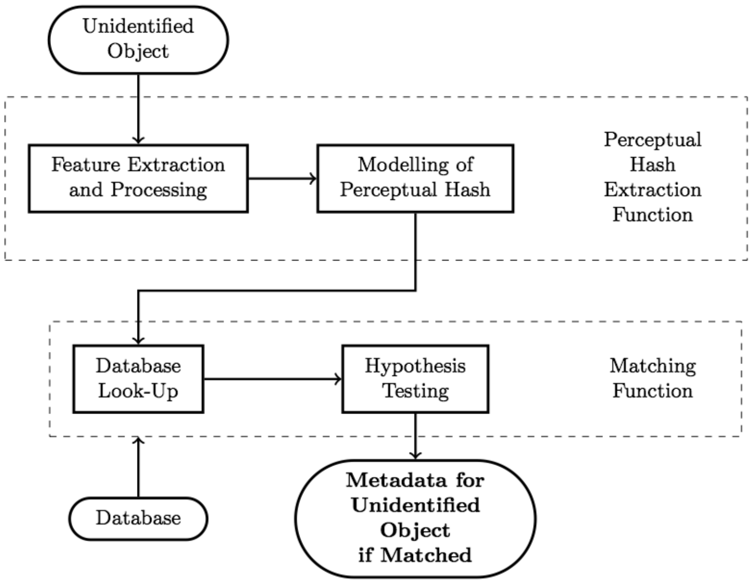

Unidentified Object
Feature Extraction and Processing
Modelling of Perceptual Hash
Perceptual Hash Extraction Function
Database Look-Up
Hypothesis Testing
Matching Function
Database
Metadata for Unidentified Object if Matched

Figure 2.11: The pHash workflow

> Block mean value based image hash function.

When a perceptual hash functions is used to authenticate media objects, even a small number of false positives is unacceptable. For an adversary, it must be impossible for any media object x to construct a perceptually different media object y such that H(x) = y. Likewise for perfect unpredictability, a equal distribution of the hash values is needed.

A problem when developing perceptual hash functions is that authentic media objects can not be precisely separated from not authentic ones. Therefore proposes a continuous interpretation of authentic: "An image which is bit by bit identical to the original image is considered completely authentic (authenticity measure of 1.0). An image which has nothing in common with the original image would be considered not authentic (authenticity measure of 0.0). All other images would be partially authentic. Partially authentic is a loosely defined concept and measurement of the authenticity is subjective, and changes from application domain to application domain."

Feature extraction and processing Normally the media content must be preprocessed in order to be processed by a perceptual hash function. In the case of an image, such required preprocessing steps can be to resize the image to a given resolution or to convert it to levels of gray.

Modelling of perceptual hash A perceptual hash is calculated using the features extracted in the previous step.

Database look-up To compare two perceptual hashes, special search algorithms and distance/similarity functions according to the used perceptual hash function must be used. The most often used are the *Bit Error Rate (BER)*, the *Hamming distance* and the *Peak of Cross Correlation (PCC)*. The first two measure the distance between two hash values, whereas the latter measures the similarity between two hash values.

Hypothesis testing Based on a pre-defined threshold it is determined if there is a match. Therefore the determination of an adequate threshold, in accordance with the actual application scenario, is critical.

Chapter 3

Approach to UI Clustering

The problem of finding visual similar application in a dataset of sample is a matter of comparing structures of application UI. The comparison should be valid without running the application, to reduce memory and time needed from the process. In order to quickly compare a new sample with the whole dataset, we can create a model to represent similar applications, using clusters. This chapter describes the approach to the clustering process. Each application is preprocessed to obtain a good representation of the UI before clustering.

In Section 3.1 we analyze how UI is developed in Android systems; in particular in 3.1.1 we describe how layouts displayed are created. In 3.1.2 we explain relations among views and in 3.1.3 we introduce the problem of mining tree structured data from layout file.

In Section 3.1.4 we describe the problem of mining frequent subtrees introducing *TreeMiner* and its variant *CMTreeMiner*.

In Section 3.2 we introduce the concept of graph clustering (3.2); we illustrate our hierarchical clustering (3.2.2) approach of how to group applications based on tree representations (3.3), using the similarity measure in Section 3.2.3.

In Section 3.4 we show how to use the created cluster to search for similar APK with respect to a new sample; implementation details can be seen in Chapter 4.

3.1 Android UI representation

In this Section we describe how developers use Android SDK to build the UI of an applications and the problems we deal with to create a structure that well represents the UI of an application. Such method will avoid Android screen fragmentation problem that we described in Section 2.1.2.

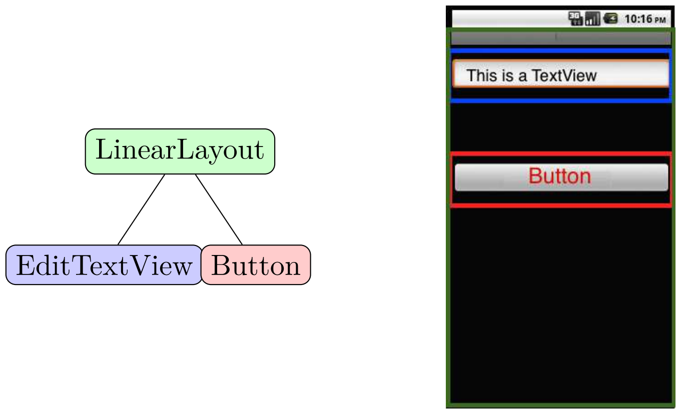


Figure 3.1: Simple layout structure

3.1.1 Layout Representation

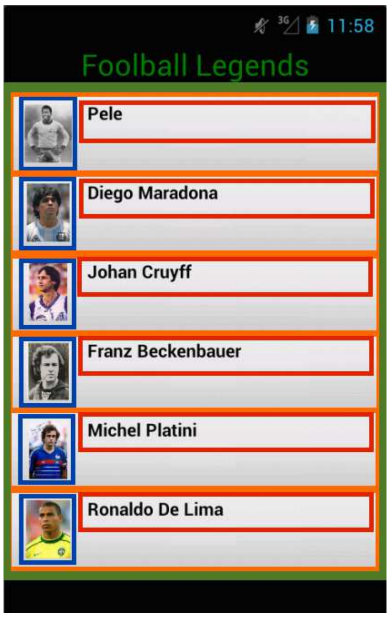
In Android applications, the UI is created by using an XML file depicting a structure of the user interface that the final user sees as a hierarchy of elements.

There are (1) elements that simply group other elements, acting like a father, and (2) elements which represent a visible object that the user can interact with. For example, to build a view which contains a field where the user can input text, with a button to confirm, the developer must create a structure with three elements (Figure 3.1): the first element will contain the other two and will be the parent. This structure will be described in the xml file corresponding to the *view*. A *view* is an object that represents contents to the user. There are multiple types of views defined in Android Software Development Kit [3]. There (1) views that represent a concrete content like text (TextView), image (ImageView) or Clock (Clock) and (2) views that group content like list (ListView) or sequence (LinearLayout).

This tree structure can be composed dynamically as the user makes use of the application. A list of elements (for example a list of conversations in a message app) can be represented by a parent object of type ListView that contains one or more elements of any kind. If the developer doesn't know a priori the amount of elements that will be shown in the UI, he can create two files of layout and combine them by code. In the application shown in Figure 3.2a, developer uses a main structure with a ListView that will contain several elements and a second structure that will be inserted once for each entry of the list. The secondary structure has no limitation in the number of elements.

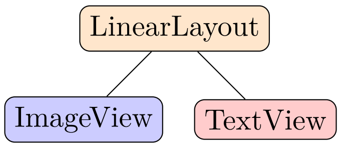
The combined view that will be shown will be the one in Figure 3.2d in case that there will be 3 elements in the list.

Sometimes the developer can also build a view without creating a XML structure, but only defining it by code. In this case, the elements of the

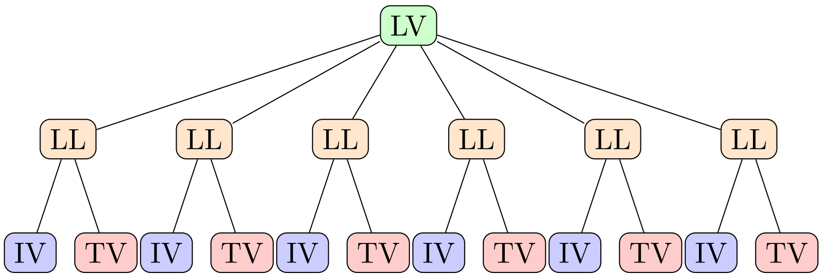


(a) ListView example

(b) Main Layout file structure



(c) Secondary Layout file structure



(d) Joined dynamic layout created during execution

Figure 3.2: Listview Usage Example

view will be written using the code. In our approach we search into the code to build the layout file that represent a code-defined view. Further details about how it can be done are described in the chapter 4.1. In our approach, an APK will contain a certain amount of files that will represent views as tree structure and that are combined together during execution of the application. Relationships among elements of the files can be found and, by manipulation of the layout files, it is possible to create a series of structures that will represent all the views of the application.

3.1.2 Hierarchies of views

Google supplies developers of an enormous amount of elements to build layouts and these elements may have several characteristics in common. The developer can define with more precision the properties and the operations that the View will have, by using an element that have general properties of another view, but some peculiar characteristics.

For example, an element as a TextView represents a text shown to the user; an EditTextView is also text field but with a particular option to be modified by the user during the execution. Also a Button is a particular TextView because in a sense it is a View that displays text (the operation associated to the button) but with the ability to be "clickable" by the user to perform an action.

The Android toolkit for developing applications defines a hierarchy of Views to represent the heredity of properties of the classes.

In our approach, views that directly inherits properties from another object will be considered as the same object, in order to compare same kind of objects.

Besides default views in the Google toolkit, developers can create their own by inferring the properties of the new class from another. For example, a new View called *MyTextView* can be created by describing it as a special *TextView*. This view will have all properties inherited from a TextView, plus other added by the developer. As a matter of fact, the new *MyTextView* can be considered as a TextView for similarity purpose (under these mentioned considerations).

With these considerations, we can represent the layouts that the application displays to the user as a sequence of tree structures.

3.1.3 Structure Analysis

In order to define a UI similarity, we can now compare the structure of the layouts of the applications. The idea behind our approach is that two applications which have a similar User Interface will have similar structure layout. For example, if we consider two messaging apps, they will have for sure:

a conversation list represented with an object *ListView* with a series of object (the conversations) each with a *TextView* (the conversation name) and an *ImageView* (the contact picture);

a conversation view represented with a *ListView* each with a *TextView* (messages in the conversation) plus a *EditTextView* (where the user can type a message) and a button (to send the message).

The more structures two applications will have in common, the more they will be considered similar. Now the problems moves to find frequent structures in dataset of trees. This problem is called *Frequent Subtrees Mining*

3.1.4 Frequent Subtree Mining

Mining frequent subtrees from database of labeled treed is a new research field that has many practical applications in areas such as computer networks, Web mining, bio-informatics, XML document mining, etc. These applications share a requirement for the more expressive power of labeled trees to capture the complex relations among data entities. Although frequent subtree mining is a more difficult task than frequent itemsets mining, most existing frequent subtree mining algorithms borrow techniques from the relatively mature association rule mining area.

Given an alphabet Σ of items and a database $\mathcal{D}$ of transactions $\mathcal{T} \subseteq \Sigma$, we say that a transaction *supports* an itemset if the itemset is a subset of the transaction. The number of transactions in the database that support an itemset S is called the *frequency* of the itemset. Given a threshold *minsup*, the frequent itemset mining problem is to find the set $\mathcal{F} \subset 2^{\Sigma}$ of all itemsets S for which $support(S) \geq minsup$.

Given a threshold *minfreq*, a class of trees $\mathcal{C}$, a transitive subtree relation $P \preceq \mathcal{T}$ between trees P, $\mathcal{T} \in \mathcal{C}$, a finite data set of trees $\mathcal{D} \subseteq \mathcal{C}$, the frequent tree mining problem is the problem of finding all trees $\mathcal{P} \subseteq \mathcal{C}$ such that no two trees in $\mathcal{P}$ are isomorphic and for all $P \in \mathcal{P} : freq(P, \mathcal{D}) = \sum_{T \in \mathcal{D}} d(P,T) \geq minfreq$, where d is an anti-monotone function such that $\forall T \in \mathcal{C} : d(P', T) \geq d(P,T)$ if $P' \preceq P$. We will always denote a *pattern tree* – a tree which is a part of the output $\mathcal{P}$ – with a P, and *text tree* – which is a member of the dataset $\mathcal{D}$– with aT. The subtree relation $\mathcal{P} \preceq T$ defines whether a tree $\mathcal{P}$ occurs in a tree T. The simplest choice for function d is given by the indicator function:

$$d(P,T) = \begin{cases} 1 & P \preceq \mathcal{T} \\ 0 & otherwise \end{cases} \tag{3.1}$$

In this simple case the frequency of a pattern tree is defined by the number of trees in the data set that contains the pattern tree. We call this frequency definition, which closely matches that of itemset frequency, a *transaction based frequency.*

TreeMiner

The *TreeMiner* algorithm developed by Zaki [25] for mining frequent ordered embedded subtrees follows the combined depth-first/breadth-first traversal idea to discover all frequent embedded subtrees from a database of rooted ordered trees. Other than the general downward closure property (i.e., all subtrees of a frequent tree are frequent), *TreeMiner* takes advantage of a useful property of the string encodings for rooted ordered trees: removing either one of the last two vertices at the end of the string encoding of a rooted ordered tree $\mathcal{P}$(with correspondent adjustment to the number of backtrack

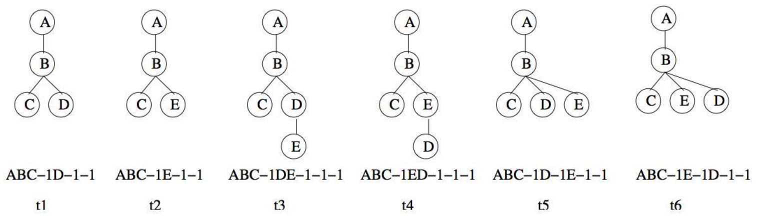


Figure 3.3: Example of joining two Rooted Trees

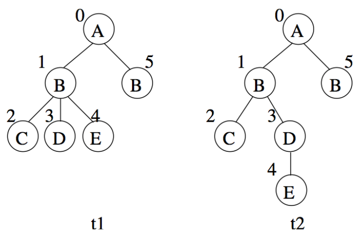


Figure 3.4: Tree Mining dataset example

symbols) will result in the string encoding of a valid embedded subtree of $\mathcal{P}$. This is illustrated in Figure 3.3. If one of the two last vertices in *t3*, *t4*, *t5* and *t6* is removed, either *t1* or *t2* results. Please note that the removal of the second-to-last vertex of *t3* yields tree *t2* as we are considering *embedded* subtrees here. Tree *t2* is not an induced subtree of t3, could not grow from *t2* when mining induced subtrees.

Figure 3.4 gives a running example of a database consisting of two transactions (with transaction-ids *t1* and *t2*, respectively). For simplicity, we assume the minimum support s=100%. 3.5 shows the part of the enumeration lattice that contains all frequent 2-subtrees with A as the prefix and all frequent 3-subtrees with AB as the prefix. From the figure we can see that the candidate generation for *TreeMiner* is very similar to the combined depth-first/breadth-first lattice traversal of frequent itemset mining.

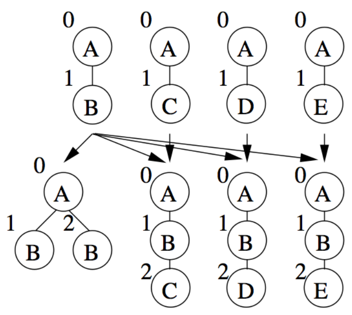


Figure 3.5: Tree Miner enumeration process

Mining maximal and closed frequent subtrees

In our approach, the tree representations of XML can considered as rooted trees; in particular, considering structural views, the order of the children doesn't matter so we chose to mine unordered trees instead of ordered ones. As consequence, we decided to use CMTreeMiner by Yun Chi [26] to retrieve frequent subtrees.

All the algorithms described discover *all* frequent subtrees from a database of labeled trees. The number of all frequent subtrees, however, can grow exponentially with the sizes of the tree-structured transactions. Two consequences follow from this exponential growth. First, the end-users will be overwhelmed by the output and have trouble to gain insights from the huge number of frequent subtrees presented to them. Second, mining algorithms may become intractable due to the exponential number of frequent subtrees. The algorithms presented by Wang et al. [27] and Xiao et al. [28] attempt to alleviate the first problem by finding and presenting to end-users only the *maximal* frequent subtrees. A *maximal* frequent subtree is a frequent subtree none of whose proper supertrees are frequent. Because they both use post-processing pruning after discovering all the frequent subtrees, these two algorithms did not solve the second problem. Later, Chi et al. [29, 26] presented an algorithm, *CMTreeMiner*, to discover all *closed* frequent subtrees and *maximal* frequent subtrees without first discovering *all* frequent subtrees. A subtree is *closed* if none of its proper supertrees has the same support as it has. Given a database $\mathcal{D}$and a support s, the set of *all* frequent subtrees $\mathcal{F}$, the set of *closed* frequent subtrees $\mathcal{C}$ and the set of *maximal* frequent subtrees $\mathcal{M}$ have the following relationship: $\mathcal{M} \subseteq \mathcal{C} \subseteq \mathcal{F}$.

To compare tree structures, we use CMTreeMiner algorithm that extracts the most frequent subtree structure in the dataset, with respect to a given minimum value of frequency, called "minimum support"; in other words the minimum support is the minimum amount of time the pattern is present in the dataset. For each subtree pattern found, the algorithm returns the subtree found and the trees, which are the applications, where the structure is present.

The algorithm searches for closed and maximal frequent subtree pattern. As from generic theory of pattern mining, we have:

> *frequent*: a pattern P is frequent if it appears an amount of time which is greater than the minimum support.

> *closed*: a pattern P is closed if there is no super-pattern (a pattern that contains P) with the same support (frequency) of P.

> *maximal*: a pattern P is maximal if none of its direct super-pattern (a pattern that immediately contains P) which is frequent.

Let's make an example. Considering the following dataset, we want to find the closed and maximal pattern with min-support 2:

1. a b
2. b c d
3. a b c d
4. a b d
5. a b c d

> *closed*: c, d, e, ac, bc, ce, de,abc, acd

> *maximal*: ce, de, abc, acd

ac, which has support 3 (it appears in 1,2 and 4) is closed because all its super-patterns {abc, acd, ace} have a support lower than **ac**; but it is not maximal, because **abc** and **acd** are frequent (even if support is lower, it is greater than minimum support).

The advantages of mining closed and maximal trees are:

> they are valuable, because they provide a compact representation of frequent itemsets;

> they form the smallest representation of frequent patterns and so they are more practical to use when a dataset is big;

> they are useful in removing redundant patterns.

As the result of the Tree Mining algorithm, we will have a list of relationships between pattern subtrees and applications.

3.2 Clustering

We decide to create groups (clusters) of applications to build a synthetic but meaningful representation of a group of applications that are similar. This will allow to:

> characterize groups of applications by patterns;

> quickly search the membership of a new sample to the dataset.

3.2.1 Graph concepts

A *labeled graph* $G = (V, E, \Sigma, L)$ consists of a *vertex* set V, an *edge* set E, an *alphabet* Σ for vertex and edge labels, and a *labeling function* $L : V \cup E \to \Sigma$ that assigns labels to vertices and edges. A graph is *directed* if each edge is an ordered pair of vertices; it is *undirected* if each edge is an unordered pair of vertices. A *path* is a list of vertices of the graph such that each pair of neighboring vertices in the list is an edge of the graph. The length of a path is defined by the number of edges in the path. A cycle is a path such that the first and the last vertices of the path are the same. A graph is *acyclic* if the graph contains no cycle. An undirected graph is *connected* if there exists at least one path between any pair of vertices, *disconnected* otherwise.

Graph clustering

Graph clustering refers to clustering of data in the form of graphs. Two distinct forms of clustering can be performed on graph data. Vertex clustering seeks to cluster the nodes of the graph into groups of densely connected regions based on either edge weights or edge distances. The second form of graph clustering treats the graphs as the objects to be clustered and clusters these objects on the basis of similarity. The second approach is often encountered in the context of structured or XML data.

Graph clustering is a form of clustering that is useful in a number of practical applications including marketing, customer segmentation, congestion detection, facility location, and XML data integration.

We have a (possibly large) number of graphs which need to be clustered based on their underlying structural behavior. This problem is challenging because of the need to match the structures of the underlying graphs and use these structures for clustering purpose. Such algorithms are discussed both in the context of classical graph data sets as well as semistructured data. In the case of semistructured data, the problem arises in the context of a large number of documents which need to be clustered on the basis of the underlying structure and attributes. It has been shown by Aggarwal, Ta, Feng, Wang, and Zaki [30] that the use of the underlying document structure leads to significantly more effective algorithms.

In our approach, the problem is to cluster *entire graphs* in a *multi-graph database*. Such situations are often encountered in the context of XML data, since each XML document can be regarded as a structural record, and it may be necessary to create clusters from a large number of such objects. We note that XML data is quite similar to graph data in terms of how the data is organized structurally. The attribute values can be treated as graph labels and the corresponding semi-structural relationship as the edges. It has been shown by Aggarwal et al. [31], Dalamagas, Cheng, Winkel, and Sellis [32], and Lian, Cheung, Mamoulis, and Yiu [33] that this structural behavior can be leveraged in order to create effective clusters. Since we are examining entire graphs in this version of the clustering problem, the problem simply boils down to that of clustering arbitrary *objects*, where the objects in this case have structural characteristics. Many of the conventional algorithms discusses by Jain and Dubes [34] (such as k-means type partition algorithms and hierarchical algorithms) can be extended to the case of graph data.

There are two main classes of conventional techniques, which have been extended to the case of structural objects. These techniques are as follows:

> *Structural distance-based approach*: this approach computes structural distances between documents and uses them in order to compute clusters of documents; for example the *XClust algorithm* [30] was designed to cluster XML schemas in order to efficient integration of document type definition (DTDs) of XML sources. It adopts the agglomerative hierarchical clustering method which starts with clusters of single DTDs and gradually merges the two most similar clusters into one larger cluster. The method by Chawathe [35] computes similarity measures based on the structural edit-distance between documents. This edit-distance is used in order to compute the distances between clusters of documents. S-GRACE is a hierarchical clustering algorithm [36]. In the work by Lian et al., an XML document is converted to a structure graph (or s-graph), and the distance between two XML documents is defined according to the number of the common element-subelement relationships, which can capture better structural similarity relationships than the tree edit-distance in some cases.

> *Structural summary-based approach*: in many cases, it is possible to create summaries from the underlying documents. These summaries are used for creating groups of documents which are similar to these summaries, in order to improve algorithmic efficiency without compromising cluster quality. The first summary-based approach for clustering XML documents was presented by Dalamagas et al. [37]. In this work, the XML documents are modeled as rooted, ordered labeled trees. A second approach for clustering XML document is presented by Aggarwal et al. [31]. This technique is a partition-based algorithm. The primary idea in this approach is to use frequent-pattern mining algorithms in order to determine the summaries of frequent structures in the data. The technique uses a k-means type approach in which each cluster center comprises a set of frequent patterns which are local to the partition for that cluster. The frequent patterns are mined using the documents assigned to a cluster center in the last iteration. The documents are then further reassigned to a cluster center based on the average similarity between the document and the newly created cluster centres from the local frequent patterns. In each iteration the document assignment and the mined frequent patterns are iteratively

reassigned until the cluster centers and the document partitions converge to a final state. It has has been shown by Aggarwal et al. That such a structural summary-based approach is significantly superior to a similarity function-based approach, as presented by Chawathe [35]. The method is also superior to the structural approach by Dalamagas et al. [37] because of its use of more robust representations of the underlying structural summaries.

Topology of trees

There are many types of trees. Here we introduce three: unrooted unordered trees (free trees), rooted unordered trees, and rooted ordered trees. In the order listed, the three types of trees have increasing topological structure.

> Free tree A *free tree* is an undirected graph that is connected and acyclic.

> Rooted unordered tree A *rooted unordered tree* is a directed acyclic graph satisfying (1) there is a distinguished vertex called the *root* that has no entering edges, (2) every other vertex has exactly one entering edge, and (3) there is a unique path from the root to every other vertex. In a rooted unordered tree, if vertex *v* is on the path from the root to vertex *w* then is an *ancestor* of *w* and *w* is a *descendant* of *v*. If, in addition, (v, w) ∈ E, then *v* is the parent of *w* and *w*is a *child* of *v*. Vertices that share the same parent are *siblings*. A vertex that has no descendant other than itself is called a *leaf*. The *depth* or *level* of a vertex is defined as the length of the path from the root to that node.

> Rooted ordered tree A *rooted ordered tree* is a rooted unordered tree that has a predefined ordering among each set of siblings. The order is implied by the left-to-right order in figures illustrating an ordered tree. So for a rooted ordered tree, we can define the *left* and the *right* siblings of a vertex, the *leftmost* and the *rightmost* child or sibling, etc.

The *size* of a tree *T*, denoted as |T|, is defined as the number of vertices the tree has. A *forest* is a set of zero or more disjoint trees.

For free trees and rooted unordered trees, we can define *isomorphisms* between two trees. Two labeled free trees T1 and T2 are isomorphic to each other if there is a one-to-one mapping from the vertices of T1 to the vertices of T2 that preserves vertex labels, edge labels, and adjacency. Isomorphism for rooted unordered trees are defined similarity except that the mapping should preserve the roots as well. An *automorphism* is an isomorphism that maps a tree to itself.

In our approach, we use to create summary of XML files as Aggarwal's by frequent subtree mining but we chose to use a hierarchical clustering algorithm (as XClust) to group applications based on the pattern found in the dataset.

3.2.2 Agglomerative Hierarchichal Clustering

We developed a Agglomerative Hierarchical Clustering (AHC) algorithm specific for our domain. Starting from a list of elements, the AHC repeats simple steps:

1. defines similarity among all elements;
2. joins the two most similar elements creating a new element;
3. deletes joined elements;
4. repeats until a certain condition is verified.

As known by Clustering theory, we also needed to define a similarity measure, a join operation and a stopping criterion for the algorithm. We choose the Jaccard similarity measure that is useful to compare two sets, that in our case are the list of patterns of an application.

3.2.3 The Jaccard Similarity

The *Jaccard Similarity Index* is a measure of similarity of sets. Let's suppose to have two sets of elements; in this case, it is calculated as:

$$\mathcal{J}(A, B) = \frac{|intersection(A, B)|}{|union(A, B)|} \tag{3.2}$$

This value is always symmetric, that means Sim(C1,C2)=Sim(C2,C1).

3.3 Clustering

We beging by creating one *cluster* for each application; each group will contain an application of the dataset. Similarity among clusters is computed considering the common patterns of the applications inside clusters.

To compute the Jaccard Index (that is, the similarity value) between two clusters, we consider as sets the list of patterns contained in the applications.

We can compute the similarity as an average of the similarity of the single applications in the cluster. Generically speaking, if a cluster A contains N apps $\{a_1, a_2, \dots, a_N\}$ ad cluster B contains M apps $\{b_1, b_2, \dots, b_M\}$ the similarity of A with B would be the average of the Jaccard Indexes computed for every combination of a_i with b_i, considering as set the patterns of a_i and the patterns of b_i.

Once we have computed all the values, we can join the most similar clusters, creating a new cluster that contains all of the applications of both the joined clusters. At each step of the algorithm, the most similar couple of clusters will be joined.

The stopping criterion has been determined empirically by looking at the decrease of the maximum similarity value. Let's consider a single step of the algorithm with a maximum similarity value among the clusters of S_i; if this value is much lower than the maximum similarity value in the previous step S_{i-1}, then the clustering operation stops and the current amount of clusterffd

To obtain a list of patterns related to each cluster, we consider the most common patterns of the applications inside.

Let's consider a cluster C with N apps $\{a_1, a_2, \ldots, a_N\}$ and each applications a_i contain patterns $P_i\{p_i1, p_i2, \ldots, p_iM\}$. The patterns representation of the cluster will be those which are in common for every combination of a_i. This operation is done by combining (by union) intersections among the sets of patterns of each application inside the cluster and can be computed during the clustering process itself.

$$\begin{aligned} pattern(C) &= \\ &= \bigcup((P_i \cap P_j)) \\ &= (P_1 \cap P_2) \cup (P_1 \cap P_3) \ldots (P_{N-1} \cap P_N) \\ &= (\{p_{11}, p_{12}, \ldots, p_{1M}\} \cap \{p_{21}, p_{22}, \ldots, p_{2M}\}) \cup \ldots \\ & \quad \cdots \cup (\{p_{(N-1)1}, p_{(N-1)2}, \ldots, p_{(N-1)M}\} \cap \{p_{N1}, p_{N2}, \ldots, p_{NM}\}) \end{aligned} \tag{3.3}$$

With this operation, a pattern representation for each cluster is obtained; a representation like this is useful to give a meaning to a cluster, and for searching (section 3.4).

3.4 Searching process

If a new application must be added to the dataset or, simply, an user wants to know which applications are the most similar to his sample, the new application must be connected to a cluster. This operation can be done without computing a similarity value with each application, but only calculating similarities with respect to clusters; in this way, it will reduce drastically the amount of time of the searching process.

Before assigning the application to a cluster, if the APK was not in the dataset before, we need to know which patterns in the dataset are contained in the application.

The approach is the same mentioned in Section 3.1.2, so we can produce a tree representation of the application UI.

Tree comparison A method for comparing two trees and check if a tree is contained in another tree is needed to know which subtrees patterns are

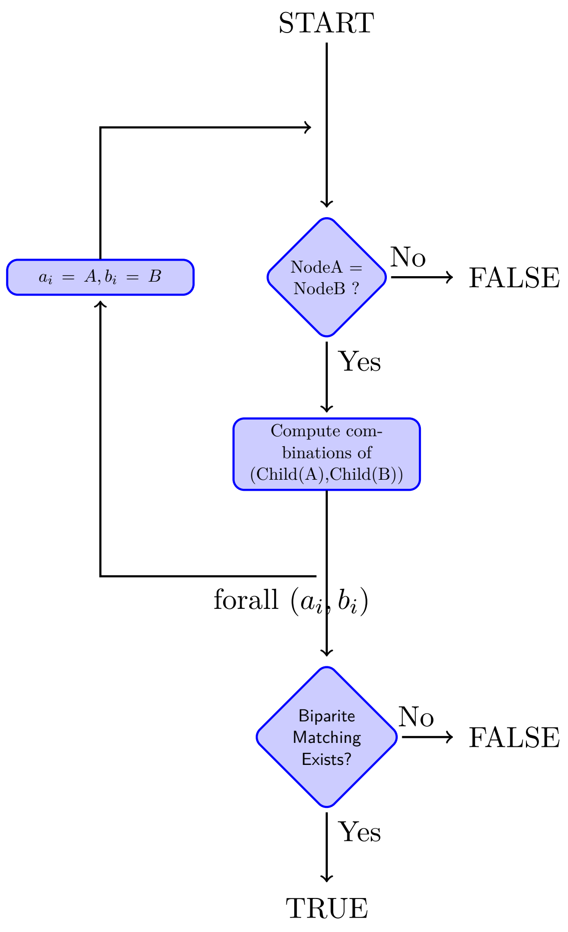


Figure 3.6: Subtree Matching Process

contained in the application tree as shown in Figure 3.6.

The algorithm works as follows. Taken into account two nodes (of two tree A and B), it checks if the nodes are equal. If the nodes are check, for the two trees to be equal also the subtrees starting from that couple of nodes must be equal. So it recall himself to check if child nodes are equal. The comparison of two subtrees must be coherent in structure, that means we have to find a correspondence between the subtrees of A with the subtrees of B with this rules.

> for every subtree a_i of A a subtree b_i of B must exist that is equal;

> for the created combinations every b_i must be used only once.

This problem is known as the *Maximum Bipartite Matching problem*: in a bipartite graph, a bipartite matching is a set of the edges chosen in such

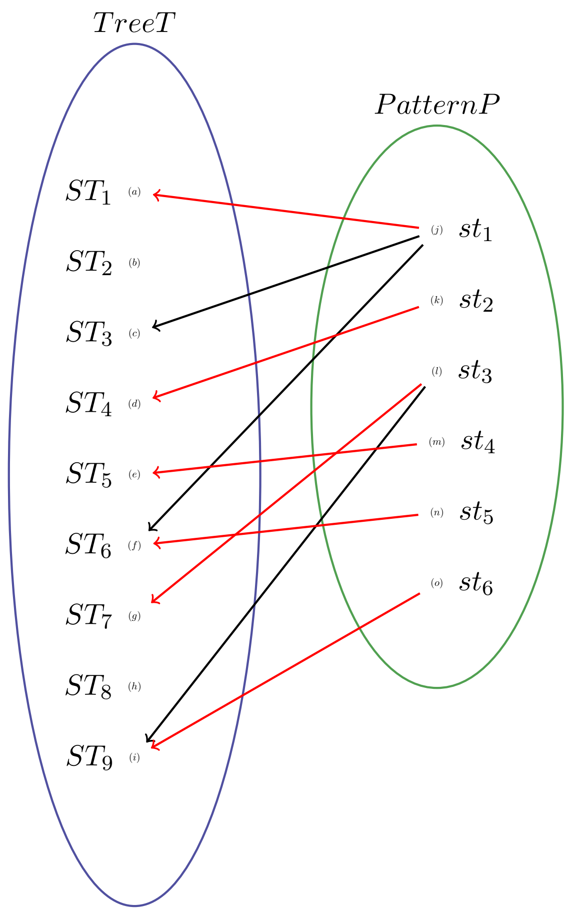


Figure 3.7: Bipartite matching example

a way that no two edges share an endpoint; a maximum matching in which the maximum number of edges possible is used. In our case, the nodes are the subtree (divided with respect to belonging) and edges are relationships among subtrees that satisfy: b_i is contained in a_i. The solution is to convert the graph into a flow network and looking for the maximum flow. See Section 4.5 and Appendix A for details.

Using tree comparison we can establish which patterns are contained in the APK and we can calculate a similarity value between an application and a clusters.

The new APK sample will be assigned to the cluster (or the clusters) that has the maximum Jaccard Index value.

At this point, we can be pretty sure, as our tests prove (see chapter 5) that the applications inside the cluster will be the most similar ones to the sample sought with respect to the whole dataset.

If we want to know which applications inside the cluster are the most similar to a new sample given, we can use the Jaccard similarity again, considering the sets of patterns that the new sample contains with respect to the patterns of each application.

The search algorithm can order the application in the cluster with respect to the Jaccard Index computed for the new app and return the N most similar applications.

Chapter 4

Implementation

This chapter describes the tool that has been defined and developed in Python. Following steps of KDD, the algorithm can be divided in 4 parts: preprocessing, transformation, treemining, clustering.

Initially, APKs are selected from an input folder that can be set by the user and are decompiled using Apktool in order to obtain resources files that represent the UI of the application and the Dalvik machine files which represent the application logic.

Obtained data are cleaned, reorganized and indexed to get a file which is a good representation of the UI flow that is presented to the normal user utilization. This operation is performed by transformation of the XAML file of Android with the aid of the *.smali* files which are created by decompilation tools.

At this point, the created files are considered as a tree and are given to the tree-mining algorithm, that returns a list of the most frequent patterns among the trees and the trees (referring to the APKs) in which the patterns where found.

The clustering algorithm developed (1) defines a measure of similarity both for patterns and applications and (2) cluster APKs based on this. This method allows to quickly search which cluster and which applications are the most similar ones without comparing the whole dataset (of applications) but comparing the frequent patterns found in the new sample with the list of patterns related of each cluster.

4.1 Preprocessing

4.1.1 APK decompilation

The system takes in input the APKs found in a given path and, for each one of them, Apktool process is launched in a separate thread, using the pool Python module. Apktool creates a folder with the name of the APK that contains all the file related to the APK.

```
apkname
    ├── AndroidManifest.xml
    ├── bin
    ├── libs
    ├── smali
    ├── asset
    └── res
          ├── drawable
          │       └── icon.png
          ├── layout
          │        ├── main.xml
          │        ├── view1.xml
          │        └── view2.xml
          ├── layout-hdpi
          │           └── view-big.xml
          └── values
                  └── strings.xml
```

Figure 4.1: APK Decompiled Structure

To decompile an apk we can invoke Apktool simply with the command:

```
Apktool d HelloWorld.apk
```

Listing 4.1: Apktool execution command

The problem who has to be taken into account is that some parts of the final application may not be obtained because of various obfuscation techniques that developer can adopt. Fortunately, obfuscation techniques often involve logic of the application and not the resources files used. So even if Dalvik files presents obfuscated names, the resources cannot contain a different identification of the layout type.

After decompilation (Figure 4.1), with respect to previous situation now we have:

AndroidManifest.xml in a readable format;

smali a folder which contains a structure of the classes.dex file in a more readable format;

bin a folder containing binary files;

lib native libraries that the application may use via NDK;

asset the asset folder;

```
apkname
    ├── ...
    ├── layout
    │       ├── main.xml
    │       ├── view1.xml
    │       └── view2.xml
    └── layout-hdpi
            └── view2.xml
                    └── ...
```

Figure 4.2: Layout folder

res folder containing all resources used by the application.

The files necessary to the system are in the *res* and *smali* folders.

Res folder contains the resources files used by the application. We can divide them in various categories:

> *drawable*: external resources (as images) that are included in the application;

> *layout*: resources that define the architecture of the UI in an Activity or a component of the UI;

> *menu*: defines the structure of the application menu that can be invoked by the user;

> *raw*: raw files as, often used for certificates;

> *values*: support files for defining array, colors, dimensions, strings and styles.

For our purpose, we can forget about all except for the layout category. After decompilation, we will find a variable number of layout folders. Usually there is always a folder called *layout* or *layout-port* which contains all the default layouts of the application (Figure 4.2). Moreover, developer can define resolution-specific layouts and landscape-specific layouts.

For example the folder layouts-hdpi will contain layouts that will be different to the default one within device with bigger resolution and the folder layouts-ldpi will contain layouts for device with smaller resolution.

4.2 Transformation

The *ElementTree* Python module supplies classes and methods to import, manipulate and export XML files as a tree structure with child and parent relationships.

Preprocessing phase of decompiled file starts with the exploration of the XML files to produce a more representative tree structure of the APK.

4.2.1 The XAML structure

Let analyze the XML files used by Android to define the layouts of the applications. They are called XAML files. There is no DTD associated with this kind of XML, because the rules that the XML files must follow are defined by the Android SDK. As we said before, using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements. Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout. For example, in Listing 4.2 we can see an XML layout that uses a vertical LinearLayout to hold a TextView and a Button.

```xml
<?xml version="1.0" encoding="utf-8"?>
    <LinearLayout xmlns:android="http://schemas.android.com
        /apk/res/android"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical" >
        <TextView android:id="@+id/text"
            android:layout_width= "wrap_content"
            android:layout_height= "wrap_content"
            android:text="Hello, I am a TextView" />
        <Button android:id=\"@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
    </LinearLayout>
```

Listing 4.2: XAML file example

When the developer compiles applications, each XML layout file is compiled into a View resource. The layout resource from the application code can be loaded in an *Activity.onCreate()* callback implementation. It can be done by calling *setContentView()*, passing it the reference to the layout resource in the form of: *R.layout.layoutfilename*. The structure of the XML will represent a hierarchy of layouts that will compose the final view presented to the final user.

In Table 4.1 there are examples of view classes that can be defined to describe a layout, similar to that one in Figure 3.2 (b).

The developer can define layout attributes (using XML attributes) as size, padding and margin to describe characteristics of the layout.

There is also a hierarchy of the class type that are defined, for example, a

Table 4.1 View classes example

TYPE	Description
AbsSpinner	An abstract base class for spinner widgets.
AutoCompleteTextView	An editable text view that shows completion suggestions automatically while the user is typing.
Button	Represents a push-button widget.
CheckBox	A checkbox is a specific type of two-states button that can be either checked or unchecked.
Chronometer	Class that implements a simple timer.
EditText	EditText is a thin veneer over TextView that configures itself to be editable.
FrameLayout	FrameLayout is designed to block out an area on the screen to display a single item.
GridLayout	A layout that places its children in a rectangular grid.
GridView	A view that shows items in two-dimensional scrolling grid.
ImageView	Displays an arbitrary image, such as an icon.
LinearLayout	A Layout that arranges its children in a single column or a single row.
ListView	A view that shows items in a vertically scrolling list.
MediaController	A view containing controls for a MediaPlayer.
NumberPicker	A widget that enables the user to select a number from a predefined range.
ProgressBar	Visual indicator of progress in some operation.
RadioButton	A radio button is a two-states button that can be either checked or unchecked.
RadioGroup	This class is used to create a multiple-exclusion scope for a set of radio buttons.
RelativeLayout	A Layout where the positions of the children can be described in relation to each other or to the parent.
ScrollView	Layout container for a view hierarchy that can be scrolled by the user, allowing it to be larger than the physical display.
SearchView	A widget that provides a user interface for the user to enter a search query and submit a request to a search provider.
TableLayout	A layout that arranges its children into rows and columns.
TextClock	Can display the current date and/or time as a formatted string.
TextView	Displays text to the user and optionally allows them to edit it.
VideoView	Displays a video file.

simple field that allows the user to insert text can be added to the layout by using *android.widget.EditText* which is subclass of *android.widget.TextView*, which, in turn, is subclass of android.widget.View.

In our approach, all the layout XML files in *layout* and *drawable* folders are joined together in a unique XML file. All the tags which refer to another layout file are substituted with the corresponding XML file nested inside the current tree and the source is deleted.

Layouts file can be nested recursively so that another layout will be considered as child of the current. This operation can be done by putting the name of the layout file as value instead of the type of the view.

Let's now consider this case: there are three identical layouts, except for one tag which is in the first case *AutocompleteEditText*, in the second case *SearchEditText*, and in the third case *EditText*. All this three classes of layout component represent three variants of *EditText*, because in the SDK both *AutocompleteEditText* and *SearchEditText* are known direct subclasses of the third. The algorithm has been build to consider always the inheritance of the class in the preprocessing phase, so that, in this case, the two tags are substituted by the tag *EditText*.

```
    <LinearLayout>
        <include>
        "./sublayout.xml"
        </include>
        <Button/>
    </LinearLayout>
```

(a) Main layout file

```
    <EditTextView/>
    <EditTextView/>
```

(b) sublayout.xml content

$\Downarrow$

```
    <LinearLayout>
    <EditTextView/>
    <EditTextView/>
    <Button/>
    </LinearLayout>
```

(c) Combined layout file

Figure 4.3: Layout inclusion transformation

4.2.2 The *smali* structure

```
.class public Lcom/test/helloworld/HelloWorldActivity;
.super Landroid/app/Activity;
.source "HelloWorldActivity.java"

# direct methods
.method public constructor ()V
    .locals 0

    .prologue

    invoke-direct {p0}, Landroid/app/Activity;->()V

    return-void
.end method

# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
    .locals 2
    .parameter "savedInstanceState"

    .prologue
```

```
22
23         invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(
               Landroid/os/Bundle;)V
24
25         new-instance v0, Landroid/widget/TextView;
26         invoke-direct {v0, p0}, Landroid/widget/TextView;-><init>(
               Landroid/content/Context;)V
27
28         .local v0, text:Landroid/widget/TextView;
29         const-string v1, "Hello World, Android"
30
31         invoke-virtual {v0, v1}, Landroid/widget/TextView;->
               setText(Ljava/lang/CharSequence;)V
32
33         invoke-virtual {p0, v0}, Lcom/test/helloworld/
               HelloWorldActivity;->setContentView(Landroid/view/View
               ;)V
34
35         return-void
36     .end method
```

Listing 4.3: A smali file example

Breaking up the sections in Listing 4.3, we have:

1. class declarations from lines 01-03;
2. a constructor method from lines 06-14;
3. a larger onCreate() method from lines 17-36.

This is an example of how to create a *TextView* by code; the *TextView* can be a layout for instance. The corresponding layout file will contain a single *TextView* element. A developer can define a personalized layout by code; in this case, in the XML file, the layout will be indicated with the reference to the code where the view has been defined, corresponding to the *.smali* file.

In our approach, we check these *smali* files to retrieve the views that was written by code and create a new layout file that represents the view. All the tags with a SMALI file link are substituted with the view defined in the corresponding file. Every field found (after checking to be a real layout) is inserted as child of this element and the element takes the name of the superclass the SMALI.

This case is valid also for external extension of the app (Facebook login, actionbarsherlok, etc. . .) that are placed inside the layouts file with link to the path of the layout file or the code where the layout is defined.

```
.class public Lcom/km/launcher/Folder;
.super Landroid/widget/LinearLayout;
.source "Folder.java"

# interfaces
.implements Lcom/km/launcher/DragSource;
.implements Landroid/widget/AdapterView$
  OnItemLongClickListener;
.implements Landroid/widget/AdapterView$OnItemClickListener;
.implements Landroid/view/View$OnClickListener;
.implements Landroid/view/View$OnLongClickListener;

# instance fields

.field protected mCloseButton:Landroid/widget/Button;

.field protected mContent:Landroid/widget/ListView;

.field protected mDragger:Lcom/km/launcher/DragController;

.field protected mLauncher:Lcom/km/launcher/Launcher;

# direct methods
...

```

(a) Example of smali file

$\Downarrow$

```
<LinearLayout>
    <Button/>
    <ListView/>
</LinearLayout>

```

Figure 4.4: Transformation of SMALI into a layout file

4.2.3 List of Views

Some layouts can be built at runtime by including another layout file (corresponding to a single XML file) multiple times.This operation is very common when developers want to represent a list of elements whose size can be modified during execution. We showed an example in Figure 3.2. An *ArrayAdapter* class must be defined and must be attached to the corresponding layout; by default, *ArrayAdapter* creates a view for each array item by calling *toString()* on each item and placing the contents in a *TextView*.

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
android.R.layout.simple_list_item_1, myStringArray);

ListView listView = (ListView) findViewById(R.id.listview);
listView.setAdapter(adapter);
```

Listing 4.4: List Adapter

In the *smali* file corresponding to the ListView object, a new-instance method can be found. In our approach we seek for *smali* files in which a list element is connected to a group of views and we attach to the corresponding element (*ListView* in this case) multiple child elements (in this case the *TextView*).

All the *ListView* with no child are searched in the SMALI files and if a corresponding invoke-direct method is found, two child are inserted in the XML files with the layout passed as argument in the method. We arbitrarily choose to add two child elements to symbolize a list of sub-elements.

At this point we will have a file that describes layout of the UI very close to that executed at runtime, without needs of executing the application.

The tree mining algorithm used now use a specific format for the tree to analyze. So before passing trees to CMTreeMiner they are transformed in a text file that will be passed as argument to the tool.

4.3 Tree Mining

4.3.1 Canonical Representations for Labeled Trees

A canonical representation is a unique way to represent a labeled tree. Especially for unordered trees and free trees, the choice for a canonical representation has far-reaching consequences on the efficiency of the total tree miner. A canonical representation facilitates functions such as comparing two trees for equality and enumeration of subtrees.

In [38, 39] Luccio et al. introduced the following recursive definition for a *pre-order string*: (1) for a rooted ordered tree T with a single vertex r,

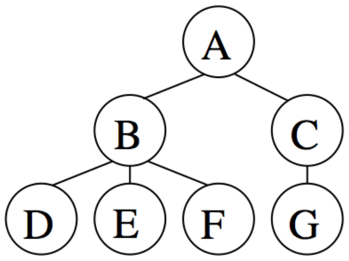


Figure 4.5: An ordered tree

the *pre-order string* of T is $S_T = l_r 0$, where l_r is the label for the single vertex r, and (2) for a rooted ordered tree T with more than one vertex, assuming the root of T is r (with label lr) and the children of r are $r_1, \ldots, r_K$ from left to right, then the pre-order string for T is $S_T = l_r S_{T_{r_1}} \ldots S_{T_{r_K}} 0$, where $S_{T_{r_1}}, \ldots, S_{T_{r_K}}$ are the pre-order strings for the bottom-up subtrees $T_{r_1}, \ldots, T_{r_K}$ rooted at $r_1, \ldots, r_K$, respectively. Luccio's pre-order strings for the rooted ordered tree in Figure 4.5 is $S_T = ABD0E0F00CG000$.

4.3.2 CMTreeMiner

CMTreeMiner can be simply called by the command:

```
CMUnorderedTreeMiner min_support trees_file output_file
```

Listing 4.5: CMTreeMiner execution command

We need to define a minimum support for the trees, that is the minimum amount of trees that must contain the subtree for a pattern to be considered valid. The format used as input file of the mining method should be a text file with the list of trees in a Node-Edges format, as described below:

```
Tree_ID_1
Number_of_Nodes
Node_label_1
Node_label_2
...
Node_label_N
Edge_label_1 Node_ID_A Node_ID_B
Edge_label_2 Node_ID_C Node_ID_D
...
Edge_label_M Node_ID_E Node_ID_F
Tree_ID_2
...
...
Tree_ID_T
```

```
...
```

Listing 4.6: Node-Edges format

The method developed creates a *trees_file.txt* which contains the XML (related to the APKs) following this conversion method.

Tree_ID is the identifier of the app;

Node_label_i is the value of the layout node (passed as a number), an appropriate structure have been created to have a relation *layout_-name-layout_ID*;

Node_ID_A/B is the identifier of the corresponding nodes in the previous list;

Edge_label_j is an hash calculated from the combination *(Node_label_-A, Node_label_B)*.

The algorithm performs with a time complexity which is logarithmic with respect to maximal frequent subtrees count. The algorithm has been slightly modified to use multiprocessing computation using *OpenMP* (http://openmp.org/). We found two main *for cycle*s and applied *OpenMP* rule with a trivial syntax.

```
#pragma omp for
for ( long s = 0; s < potential.size(); s++ )
{
  if ( potential[s].support >= threshold )
  {
    DO SOMETHING
  }
}
```

Listing 4.7: OpenMP example

Then we needed to compile the algorithm with a compatible version of *gcc* using -fomp option.

In the default configuration, CMTreeMiner returns as output only the amount of frequent subtrees found in data. So we modified the algorithm to be compliant with our software to return a structure of the frequent patterns found in a *patterns.txt* file.

```
Tid: 0
Vertices Number: 4
2 3:−−4_5 −−3_2 −−2_1
```

```
patterns.txt
    ├── Tid
    ├── Vertices Number
    ├── Edges List
    │       ├── Node label
    │       ├── Child count
    │       ├── Child ID
    │       ├── Edge label
    │       └── ... ...
    ├── automorphism
    ├── canonical string
    ├── application id list
    └── support
```

Figure 4.6: Pattern file returned from tree miner

```
9 1:--1_1
2 1:--1_2
2 1:--1_5
automorphism: 1 2 3 4
canonical string: 4 0 2 1 9 30001 2 2 30001 5 2 30002
id list: 2 4 8 10 10 12 15 16 17
support is: 9
```

Listing 4.8: Subtree Format

In the listing there is an example of *patterns.txt* file returned. As we can see we have all informations about maximal frequent subtrees.

4.4 Clustering

4.4.1 Class Structures

Two main classes are built by the system: *application* and *pattern*. The application class contains all informations about an APK file:

Table 4.2 APK class structure

Name	Description
id	identification of the APK, used for clustering
filename	the source file name
package	the package name found in manifest file
tree	ElementTree class representation of the resources
patterns	list of subtrees id found into the APK
XMLfile	path to the XML file created in previous steps

Table 4.3 Pattern class structure

Name	Description
Id	identification of the pattern, used for clustering
nodes number	amount of nodes of the tree
support	support count of the tree
apps	list of ids of application classes that contain the subtree
file path	path to the XML file that describe the subtree

The pattern class contains all informations about the mined frequent subtrees:

To cluster apps, the system only needs the list of application classes created (in preprocessing phase) and the list of pattern classes created (in transformation phase). There are three assertions that the system checks:

$$\forall patternP, a \in P.apps \implies \exists applicationA \wedge A.id = a \wedge P.id \in A.patterns \tag{4.1}$$

$$\forall appA, p \in A.patterns \implies \exists patternP \wedge P.id = p \wedge A.id \in P.apps \tag{4.2}$$

$$\forall patternP, |p.apps| = P.support \tag{4.3}$$

The first two assertions can be explained because logically there is a N-N (N to N) relationship between apps and patterns.

For both classes, an object serialization has been defined using Python pickle module. The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. *Pickling* is the process whereby a Python object hierarchy is converted into a byte stream, and *unpickling* is the inverse operation, whereby a byte stream is converted back into an object hierarchy. We don't need to save the entire tree representation of APKs by converting the *ElementTree* class into an XML file during pickling process.

4.4.2 Agglomerative Hierarchical Clustering

This phase has been build to be executed in parallel using the good Python module for multi-threading *pool.* This module supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the *Global Interpreter Lock* by using subprocesses instead of threads. Due to this, the multiprocessing module allows to fully leverage multiple processors on a given machine. It runs on both Unix and Windows machines. We implemented a simple typical Agglomerative Clustering, defining a similarity measure, a join operation and a stopping criteria for the algorithm.

Initial clusters are created from the applications and joined using common patterns in the similarity measure; as consequence, the number of initial clusters is equal to the number of application processed. The similarity value between two clusters C1 and C2 has been computed using the McQuitty method of distance [34].

```
for app1 in cluster1.apps as C1:
  for app2 in cluster2.apps as C2:
    jaccard_values <− calculate_jaccard_similarity(app1.patterns,app2.patterns):
  return avg (jaccard_values)
```

Listing 4.9: McQuitty Cluster distance

A list of Jaccard values is computed and the average value is returned depending on the chosen linkage option.

The initial similarity matrix is a NxN square matrix where N is the number of applications. This matrix is symmetric, because similarity computation is symmetric.

At each step of a Hierarchical clustering, the algorithm performs:

1. the two most similar clusters are joined together;
2. joined clusters are removed from similarity matrix;
3. new cluster is added to the matrix;
4. similarity measure of new cluster with respect to the others is computed.

The operation of joining two clusters creates a new cluster with:

```
  C.apps= sum(C1.apps, C2.apps)}
```

Listing 4.10: Direct clustering join operation

The stopping criterion has been determined by tests by looking the decrease of maximum similarity value, detail in section 5.1.1; when the stopping criterion verifies, algorithm stops and returns a set of clusters.

4.5 Search

Class representation of a cluster, at the end of AHC process contains:

apps list of application contained

patterns list of most frequent pattern of the cluster

source link to the couple of cluster that originated this cluster

The searching process allows to quickly associate a new sample to a cluster and get a list of most similar apps without checking the whole database. When a new APK is given to the searching algorithm, the preprocessing and transformation phases are performed on the sample: the application is decompiled, resources and SMALI files are extracted and elaborated and application class is created, including the *ElementTree* class representation of the UI. To find the most similar cluster we simply need to know which frequent subtrees are present in the tree representation of the application. We have developed a method *searchsubtree* which compares two trees and determines if one is included in the second.

```
subtreematch(NodeA,NodeB)
  if (NodeA != NodeB) then
    return False
  end if
  for all a in NodeA.children do
    for all b in NodeB.children do
      subtreematch (a,b)
    end for
  end for
  if bipartiteMatching (NodeA.children,NodeB.children) then
    return True
  else
    return False
  end if
```

Listing 4.11: Subtree Match algorithm pseudocode

The algorithm uses a parallel computing technique to separate branches of the seek process among threads, using the pool Python module.

Let's analyze the part of the branches association: let's suppose to have a tree T1 and a tree T2 and we have to check whether T2 is contained in T1. T1 has four child branches B1 {A,B,C,D}; T2 has four child branches B2 {E,F,G,H}.

The algorithm performs these operations:

1. checks if parent node of T1 and T2 is equal;

2. for each combination of *(child of T1, child of T2)* calls recursively to check which branches of T2 are contained in the branches of T1;

3. finds a combination of B1={A,B,C,D} with B2={E,F,G,H} so that for each branch in B2 there is an element of B1 that contains the branch;

4. if all the conditions are satisfied, returns True.

The third operation is called *Unweighted maximum bipartite matching problem*, which is a well known Operation Research problem. Finding a *maximum bipartite matching* in a bipartite graph G=(V=(X,Y),E) is perhaps the simplest problem.

The *Augmenting path algorithm* finds it by finding an augmenting path from each $x \in X$ to Y and adding it to the matching if it exists. As each path can be found in O(E) time, the running time is O(V E). This solution is equivalent to adding a super source s with edges to all vertices in X, and a super sink t with edges from all vertices in Y, and finding a maximal flow from s to t. All edges with flow from X to Y then constitute a maximum matching.

In our implementation, we decided to use the *Hopcroft–Karp algorithm*, which is an improvement of the *Augmenting path algorithm* and runs in $O(\sqrt{V}E)$ time. For detail about the algorithm check the appendix.

The final application class will contain the list of the patterns contained (that verify subtree match method).

In order to find the most similar cluster, we use the same formula applied in *clustering using pattern clusters*:

```
Cluster(App) = C argmax(calculate_jaccard_similarity(app.patterns,C.patterns))
      for C in Clusters
```

Listing 4.12: Cluster association

Moreover, the algorithm orders applications in the cluster following the Jaccard similarity value of two applications:

```
sorted( Cluster.apps,key=lambda x: calculate_jaccard_similarity(App.patterns,x
      .patterns) )
```

Listing 4.13: Sorting operation

Both operations can be performed in parallel and are computed in linear time O (N).

4.6 Request manager

We developed a class that can receive requests and perform operations described above. The Python module bind himself to a specific port and accept

TLS over SSL connection.. The request and response is simply an exchange of messages that both the server (*SimDroid-UI*) and the client (the user) must know in advance. In this scenario requests become:

ID	operation	Description	Parameters
1	clustering	clusters data	dataset-path
2	get clusters	return last created clusters	
3	search	search a new sample	APK path

Chapter 5

Test results

In 5.1 we define how test were performed using the *Request Manager* module created. In particular, in Section 5.1.2 we will describe different tests that have been on the dataset of 5.1. In 5.2 we will show and comment the results obtained using our clustering algorithm; we test our clusters using three metrics. In Section 5.2.1 we compare our clusters to the associations of a human thought; in Section 5.2.2 we evaluate similarity of the clusters with respect to the similarity of the whole dataset; in Section 5.2.3 we use Puppetdroid's rerun test to obtain a measure of effectiveness of our algorithm. In 5.3 we will compare our algorithm search process with Phash 2.3.2 and Androsim 2.3.1; in particular we will focus the comparison of views that are recovered 5.3.3 and execution time of the runs 5.3.4. In 5.4 we summarize the results obtained in the tests.

5.1 Test environment

In this section we will define the methodologies used to test our algorithm. This will bring to have different prospective of the advantages of our research.

Request manager We used the Request Manager process described in Section 4.6 to receive requests of both clustering and search. Multiple clustering request were sent to the process to obtain various configuration of clusters with respect to *min-support* parameter; for a subset of the dataset, multiple search requests were sent, to obtain a list of most similar apps to each singularly-given application.

Dataset The dataset is composed of 120 samples of applications, that differs for *categories*, *tyoe*, *sources* and *size*. We took applications that goes from 120Kb to 80Mb from both official and unofficial market (2.1.3). Applications belong to different categories with respect to Google Play 2.1.3

separation (entertainment, productivity, social, etc.) and in terms of kind of usage of the final user (messaging, calendar, camera, etc.).

Similarity matrix In order to compare both clusters and applications similarity, we built an application similarity matrix for each algorithms considered. This matrix is an NxN matrix where N is the number of sample analyzed. Note that Androsim [40] uses an asymmetric similarity measure and as consequence, the similarity matrix itself is not symmetric. Table 5.2 shows an example of similarity matrix calculated for a subset of the real dataset.

Table 5.2 Similarity matrix example among applications

Sim	app001	app002	app003	app004		app117	app118	app119	app120
app001	1	0.698	0.023	0.643		0.432	0.221	0.302	0.341
app002	0.324	1	0.652	0.834		0.456	0.673	0	0.207
app003	0.249	0.531	1	0.222		0.114	0.124	0.32	0.459
app004	0.764	0	0.43	1		0.018	0.883	0.344	0.133
app117	0.543	0.435	0.532	0.492		1	0.07	0.221	0
app118	0.298	0.719	0	0.045		0.866	1	0.348	0
app119	0.823	0.403	0.535	0.538		0.563	0.494	1	0.244
app120	0.089	0.641	0	0.18		0.547	0	0.32	1

Puppetdroid integration To have a real idea of view element and subtree relations among a couple of application, we made use of Puppetdroid (2.2). The tool tries to execute stimulation traces by seeking for views path into a second application with respect to that in the first tested one. We think that this can be a good measure for real similarity comparison. This project relies a lot on application visual similarity, in terms of real visualized structure (composition of views), so if its test found few views in a compared application, this means that the two applications are unlikely to have a similar UI structure.

5.1.1 Parameter tuning

The developed algorithm needed some parameters to be tuned, *Minimum support* and *Stopping Criterion* for clustering

Minimum support We need to define a minimum support for the trees, that is the minimum amount of trees that must contain the subtree for a pattern to be considered valid. If this value is too small, frequent subtrees returned will contain a large number of patterns of low significance. If the value is too large, Tree-Miner algorithm will take away valid patterns because of low support.

We made a lot of experiments to define the minimum support value in relation with the amount of apps to be clustered. In Figure 5.1 we plot

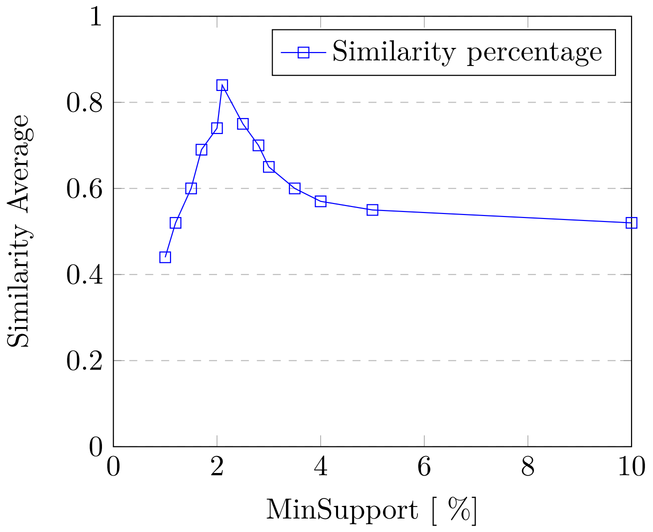


Figure 5.1: Intra-cluster similarity with respect to minimum support

the average similarity value under minsupport percentage (with respect to dataset size) used for tree mining. As we can see, we have a maximum value in between 2% and 2,5%, so we can assume that a reasonable value should be between 1/40 and 1/50 of the amount of APKs considered.

Stopping criterion The stopping criterion value as been defined by experiment. We notice that at a certain point, the maximum similarity value of the clusters decreases radically. By experiment we find a minimum $\delta MaxSim$ for which the clusters can be considered optimal. From that point onwards, the Jaccard similarity values among clusters become meaningless. As shown in Figure 5.2, the $\delta MaxSim$ there is a point, between the 90th and 100th step of the algorithm, for which the maximum similarity value has a strong drop.

When can infer that, if maximum similarity value is 40% lower than the one in the previous step, the clustering process can be stopped.

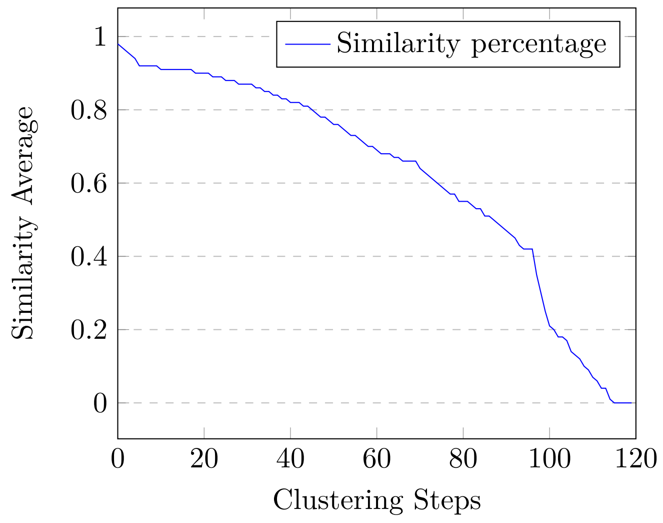


Figure 5.2: Maximum similarity during execution

5.1.2 Methodologies

We developed different methodologies for testing our clustering algorithm, focusing on evaluate both the meaning (clusters division can be understood by human) and the effective UI similarity computed.

We can divide our evaluation in four distinct tests:

> *Human evaluation*: cluster created are related with respect to groups that a human would create based on its kind of usage of the application

> *Similarities*: similarities among clusters is coherent with respect to application similarity, so that looking for a cluster instead of comparing the whole dataset of application will give nevertheless the most similar applications.

> *Views comparison*: The similarity value will be effectively a measure of structural layout comparison, that means that more applications are similar, much more the layouts structures of one can be recovered in the other (in terms of path to retrieve a view from layout root).

> *Execution Time*: The clustering process based on tree structure analysis should bring a much more fast search execution with respect to comparing all APK sample in the dataset.

5.2 Algorithm Results

In this section we will present the results produced by our approach in the various tests that have been listed in the previous section (5.1.2).

5.2.1 Human evaluation

We tried to find a human comprehensible meaning to created clusters. To evaluate this aspect, we asked some users to associate a type applications based on the really use he makes (limited to at 12 groups at max).

Type	Amount
calendar	6
clock/alarm	7
list-management	9
launcher	5
game	9
multimedia	10
news	10
messaging	12
social	13
utilities	16
file explorer	10
file editor	13

The test has the aim to verify that if two applications are of the same type (given from the user), they must belong to the same cluster. This is called a classification test based on unsupervised clustering. We can compute precision P and accuracy A of our approach as:

$$\begin{cases} P = \frac{TP}{TP+FP} \\ A = \frac{TP+TN}{TP+TN+FP+FN} \end{cases} \tag{5.1}$$

where:

> TP = applications pair correctly in the same cluster

> TN = applications pair correctly not in the same cluster

> FP = applications pair wrongly in the same cluster

> FN = applications pair wrongly not in the same cluster

Table 5.3 Confusion matrix

	Pred-YES	Pred-NO	7140
Real-YES	TP = 408	FN = 1113	1521
Real-NO	FP = 187	TN = 5432	5619
7140	595	6545	7140

In the table 5.3 we depicted the confusion matrix. Sample are assigned quite correctly and in table present both high TP and TN value then low FP and FN. The computed precision value is 0,685 and the relative accuracy is

0,817. This means that our clusters tend to group together apps that offer to the final user the same kind of usage. The explanation to this phenomenon is that, in general, apps that have similar tasks must have similar UI. This bring a developer to define a UI of his application leading from other similar-task applications.

5.2.2 Similarities

The aim of this test if to check if applications into the same cluster, at the end of the clustering process, are much more similar with respect to applications outward the cluster. This would prove that searching by computing cluster-to-application similarity will bring to find out almost the same similar applications in comparison with computing application-to-application similarity.

Table 5.5 shows the average of the similarities among applications for couple of clusters.

$$AverageSim_{ij} = \frac{\sum_{(a,b)a \in C_i, b \in C_j} Sim(a,b)}{|(a,b)a \in C_i, b \in C_j|} \tag{5.2}$$

Table 5.5 Similarity matrix example among clusters

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
c1	0.621	0.299	0.100	0.182	0.149	0.191	0.153	0.119	0.064	0.099
c2	0.166	0.557	0.242	0.148	0.091	0.144	0.107	0.310	0.343	0.055
c3	0.064	0.207	0.608	0.033	0.252	0.070	0.255	0.051	0.179	0.305
c4	0.185	0.170	0.219	0.734	0.243	0.252	0.109	0.151	0.163	0.223
c5	0.254	0.143	0.266	0.105	0.634	0.160	0.319	0.200	0.271	0.232
c6	0.270	0.168	0.348	0.174	0.188	0.606	0.181	0.251	0.116	0.261
c7	0.194	0.141	0.210	0.233	0.352	0.136	0.720	0.049	0.297	0.308
c8	0.205	0.151	0.084	0.106	0.119	0.355	0.167	0.584	0.338	0.318
c9	0.051	0.200	0.063	0.095	0.283	0.133	0.223	0.181	0.659	0.235
c10	0.230	0.133	0.242	0.217	0.262	0.225	0.067	0.334	0.180	0.670

We can see that the matrix is symmetric (for the symmetric definition of the similarity of two clusters check Section 4.4.2) and diagonal element are much higher with respect to the other elements. That table 5.5 proves that in our system we will have a much bigger intra-cluster similarity and a much lower extra-cluster similarity, as shown in plot in Figure 5.3 and 5.4. Much more the diagonal shape will be sharp, much more the clusters will be separated.

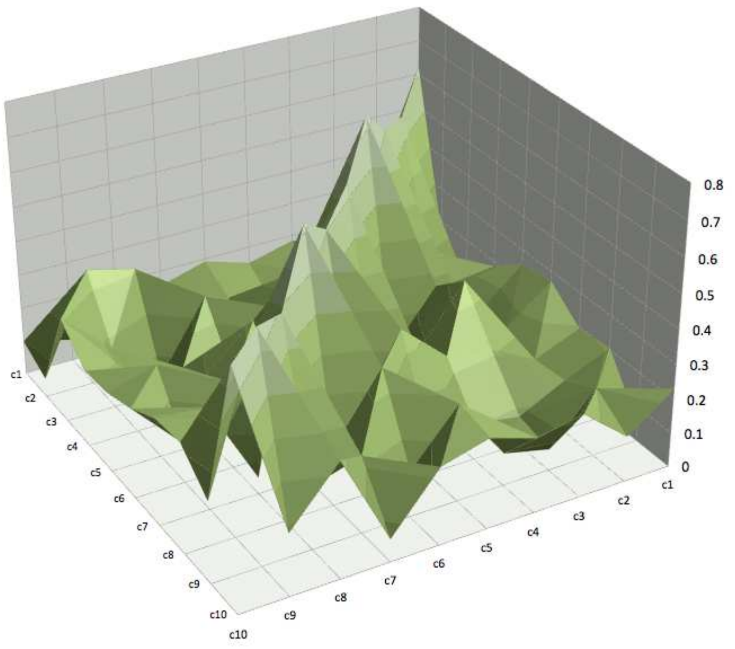
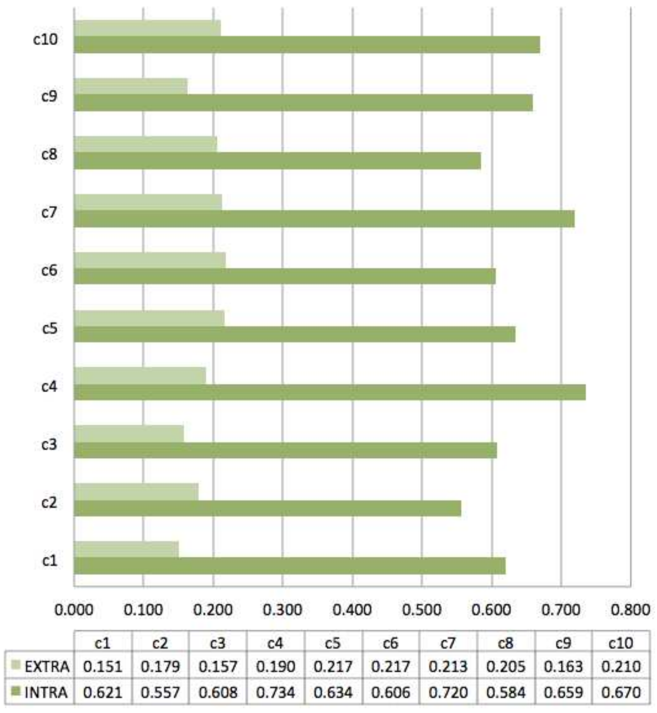


Figure 5.3: 3D graph of cluster similarity



	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
EXTRA	0.151	0.179	0.157	0.190	0.217	0.217	0.213	0.205	0.163	0.210
INTRA	0.621	0.557	0.608	0.734	0.634	0.606	0.720	0.584	0.659	0.670

Figure 5.4: Bar graph of cluster similarity

5.2.3 Consumed views

The aim of this test is to check the effective structural similarity of two applications considered similar. We expect to have an approximate dependence between the similarity value and the views that Puppetdroid succeeded in layout recover during execution. If two applications present to the user similar layouts during executions, then Puppetdroid can find the same structural path starting from the layout root up to the view element were the action is consumed (for example where the click of a button make effect on a single element). We created a subset of the dataset with 40 APK and for every application in the subset we ran a "Manual Puppetdroid test" [21] to let the program register user input (focused in stimulating views). Then we computed the application permutations in the subset (which for Puppetdroid are not symmetric) which brings to 40*39 = 1560 permutations. For each permutation (A,B) we ran a "Puppetdroid rerun test of B using A inputs" to test the input registered in application A under the execution of application B.

Table 5.7 Consumed Views among clusters

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
c1	69.67	29.92	10.83	18.24	15.45	20.54	18.34	10.64	4.10	8.30
c2	18.66	64.42	23.06	15.36	8.47	10.14	2.06	31.16	34.01	6.65
c3	4.11	17.63	72.92	4.01	25.48	6.05	19.84	4.91	15.44	34.23
c4	14.94	12.02	26.29	78.53	27.95	22.99	10.64	7.26	8.78	20.80
c5	24.53	8.72	29.53	6.61	71.30	15.60	37.12	19.15	32.52	19.43
c6	32.39	9.39	33.31	17.30	20.13	69.12	19.28	24.17	12.74	28.96
c7	17.30	9.76	22.80	18.37	35.06	16.38	79.17	5.87	26.00	29.80
c8	13.85	14.48	5.33	10.35	13.09	33.04	18.82	64.13	29.77	29.79
c9	3.78	22.78	2.76	7.84	33.99	5.17	18.35	14.56	74.24	21.01
c10	20.39	15.98	18.25	21.23	31.38	17.46	4.47	34.03	10.86	73.22

Table 5.7 shows for each cluster the average of views consumed in execution divided by the case the two applications tested where in the same cluster or not. The element in the first column represent:

$$IntraClusterViews(i) = Avg(Views(a, b)), \{a, b\}, a \neq b \wedge a, b \in C_i \quad (5.3)$$

While for the second column:

$$ExtraClusterViews(i) = Avg(Views(a, b)), \{a, b\}, a \neq b \wedge a \in C_i \wedge b \notin C_i \quad (5.4)$$

As displayed in Figures 5.5 and 5.6, for each cluster created, the views that are correctly consumed are much more with respect to the views that fails to be located.

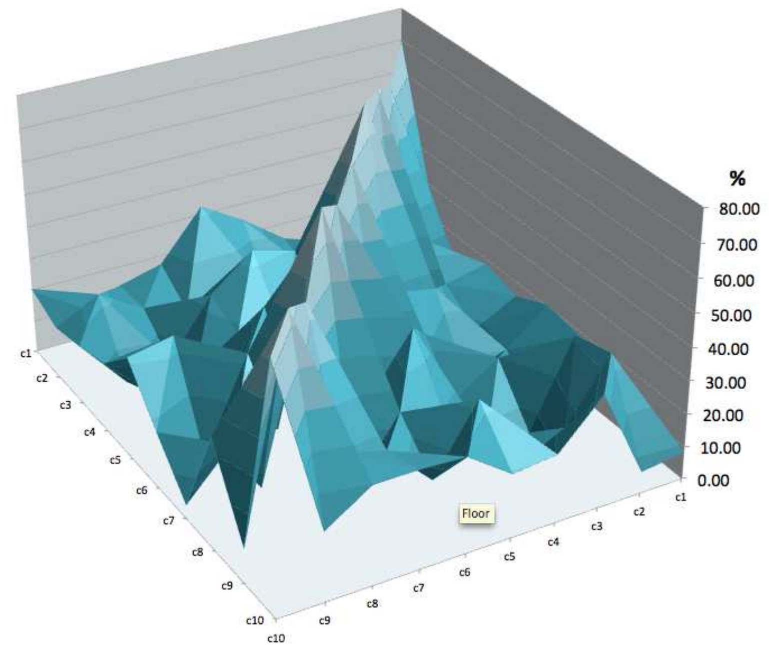
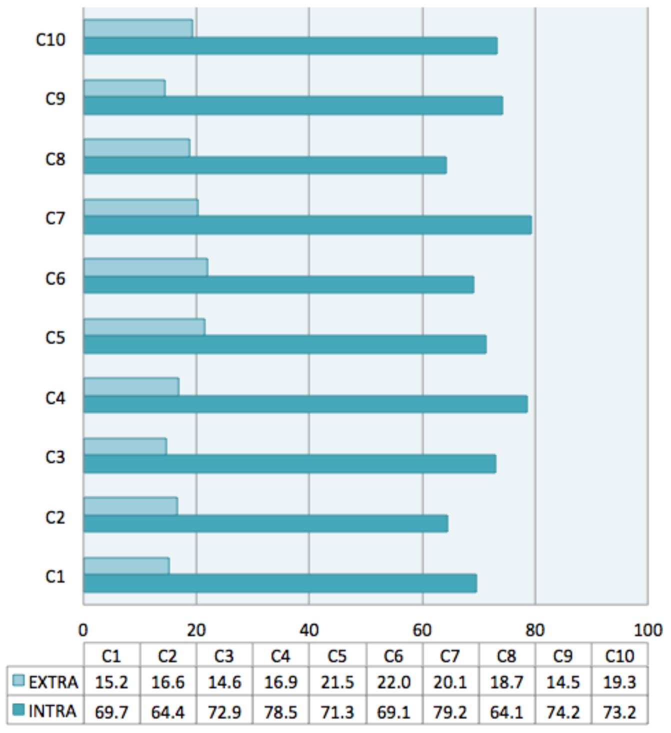


Figure 5.5: 3D graph of consumed views between clusters



	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
▢ EXTRA	15.2	16.6	14.6	16.9	21.5	22.0	20.1	18.7	14.5	19.3
■ INTRA	69.7	64.4	72.9	78.5	71.3	69.1	79.2	64.1	74.2	73.2

Figure 5.6: Bar graph of consumed views between clusters

5.3 Comparison of algorithms

5.3.1 introduction

We decided to compare our algorithm with two different Android similarity computation algorithm: Androsim (2.3.1) and Phash (2.3.2). This two algorithms are great differences in terms of methodology that strongly influence this comparison; we can enumerate these differences in:

1. PHash need to run the applications (in an emulator) while androsim need to decompile the APK
2. PHash focuses on visual appearance while androsim focuses on methods
3. PHash similarity function is symmetric while androsim's is not

We expect PHash to be the slowest because of the need of running a emulator (and mean in terms of resources) while Androsim to be inefficient while applications presents similar UI but extremely different code.

Similarity matrices To compare algorithms we computed a similarity matrix for each algorithm. These matrices are NxN matrices with N number of application in the dataset (which contains 120 sample). As result every matrix will show 1400 elements (every combination of application). Matrices are similar to that one in Table 5.2 Note that both *SimDroid-UI* and PH ash similarity matrices are symmetric while Androsim's is not.

5.3.2 Validation strategy

The validation method applied is a k-fold cross validation strategy ([41]) with k = 10. The cross validation process used start by dividing the dataset into k parts (10 in our case, that produces subsets of 12 applications). A brief explanation on how the validation is performed using this k groups is the following:

1. remove a subset from the dataset
2. build model (that means build application similarity) with new dataset
3. evaluate using removed subset
4. add the subset in the dataset again
5. repeat until all k subset are used

This process tries to avoid overfitting problem that may caused by using same samples for the model building (clustering) and for the testing.

For our domain the steps become:

1. remove a subset of 12 applications from the dataset
2. construct similarity matrix with new dataset of APKs
3. using the 12 removed APKs, evaluate searching process for each APKs, analyzing most similar applications returned with respect to visual correspondence (by Puppetdroid) and execution time
4. add the 12-APKs subset in the dataset again
5. repeat until all 120 APKs are used

In point 3, for every application in the dataset a search process three searching process are launched, one for each algorithm; the total amount of searching process is 120*3 = 360.

Searching process For our approach, the searching process involve using the method described in section 4.5. For PHash and Androsim we extrapolates the most similar apps returned by the correspondent process.

Three "Speacial Search" were acted executed too with three new type of APKs constructed as follows. Starting from an APK, new APKs are built by modifying source code:

1. an APK variant with .DEX file strongly modified - classes and methods are modified but UI representation remains the same; called **variant-DEX**
2. an APK variant with Resources file strongly modified - layouts are modified but no changes in the code were applied; called **variantRES**
3. an APK variant with both DEX and Resources file modified; called **variantBOTH**

5.3.3 Views

As in section 5.2.3, we make use of Puppetdroid to display a measure of goodness of most similar APKs found with respect to one new sample given.

We decided to evaluate the average of the percentages of Consumed Views of the 5 most similar APKs.

$$\begin{aligned}&Goodness(algorithm_i, app_j) = Avg\left(\frac{CV(app_j, app_k)}{CV(app_j, app_k) + NCV(app_j, app_k)}\right)\\&\forall app_j \in Most5Sim(algorithm_i, app_j)\end{aligned} \tag{5.5}$$

Let's try to explain this computation considering our *SimDroid-UI* . For the app A the process follows this steps:

1. find the 5 most similar APKs wrt A using *SimDroid-UI*

2. for each APK r_i returned:

    (a) launch Puppetdroid to test APK r_i using input of APK A

    (b) get the amount Consumed Views and the amount of Not Consumed Views (views which haven't been found again in r_i

    (c) compute the percentage of CV with respect to (CV+NCV)

3. average results

The results of this tests are shown in table 5.9 and plot in Figure 5.7.

Table 5.9 Consumed views algorithm comparison

	SimDroidUI	pHash	Androsim
apk1	77	51	25
apk2	87	35	48
apk3	55	60	16
apk4	73	69	25
apk5	95	10	27
apk6	90	56	20
apk7	82	10	33
apk8	94	27	47
apk9	39	49	40
apk10	36	40	52
apk11	61	20	22
apk12	31	22	43
...	...	...	...
apk115	90	16	6
apk116	34	19	29
apk117	60	48	23
apk118	94	31	17
apk119	63	67	7
apk120	86	34	3
VariantDEX	78	54	22
VariantRES	54	43	4
VariantBOTH	54	13	12

We can notice that the percentage of views found again in the APKs chosen by our *SimDroid-UI* are nearly always greater with respect to that returned by androsim and phash and much greater in average. As already demonstrated in literature ([21]), phash brings to better results with respect to androsim.

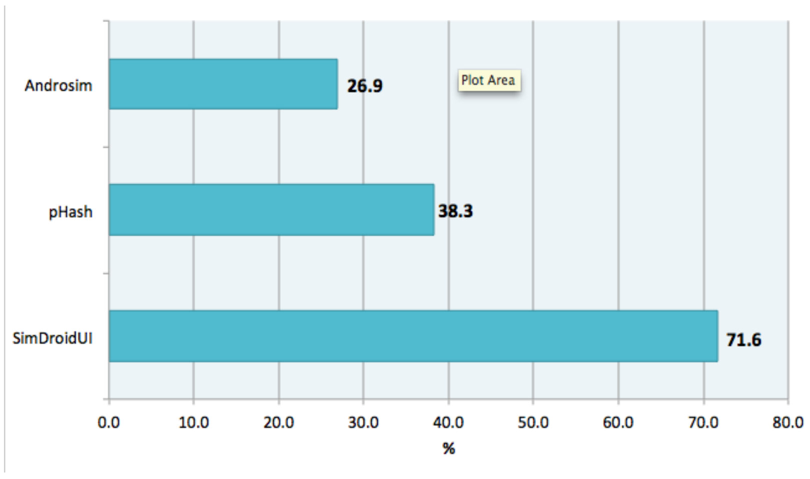


Figure 5.7: Consumed views algorithm comparison

Special APKs

Analyzing the special APKs search results (Figure 5.9) we had:

> for variantDEX androsim give worse result while PHash and *SimDroid-UI* still gives good APKs

> for variantRES androsim give best results as it would be expected; then *SimDroid-UI* shows to be more robust with respect to PHash.

> for variantBOTH all the three algorithms properly return random results.

5.3.4 Execution time

We will compare the execution time of the three algorithms in both computing a similarity measure between two sample and the action to search a new sample similarity with respect to an already built database. The graph in Figure 5.8 shows the similarity computation time with respect to the increase of sample size (table 5.11).

The long execution times and the high escalation with respect to file size of PHash is caused by the need of the algorithm to run the application and taking the first screen, that means it must run an emulator, install the APK and start the application that is much more influenced by the file size. *SimDroid-UI* is quite influenced by the size of APK because it is influenced only by the size of the layouts file,that we approximate to have always the same ratio of code size (comparing different APK size). In real applications, as much the APK size is bigger, much more the ratio between code size and layouts file size will be bigger. On the other hand, because we compared graph structured data, as we said in section 3.2.1, the complexity is quadratic order with respect to number of nodes, that we expect to increase along with APK size. Androsim is quite faster with respect to our *SimDroid-UI* because it uses a feature-hash comparison when hashes are computed from string value of the methods.

The graph in Figure 5.9 shows the search process execution time with respect to the increase of dataset size.

We can notice that *SimDroid-UI* is quite linear because of the cluster method we have applied, searching by cluster instead of scanning all applications reduce drastically time for comparing a new sample, even if the dataset increase in size. With PHash method, because it implemented a DBScan clustering algorithm, searching process is quite faster and does not scale too much with respect to dataset size; the only problem is the constant time it needs to execute the new sample. Androsim, because of it does not implement a clustering method but compare each a new sample with the entire dataset, its searching process is linear with respect to the database size.

5.4 Conclusions

We proved that our clusters are quite good in terms of grouping applications with the same scope to the end user by comparing the semantic grouping by human being with our clusters and get quite good values of precision and accuracy. By comparing inter with extra cluster similarity, we have proved that our cluster representation using patterns is good with respect to similarity of application contained inside a cluster. By comparing Puppetdroid consumed views, we proved that *SimDroid-UI* returns applications very good in terms of visual similarity; the application returned contains lot

Table 5.11 Execution time comparison of similarity computation

	SimDroidUI	pHash	Androsim
apk1	18	77	45
apk2	3	66	20
apk3	10	73	24
apk4	8	123	48
apk5	20	95	17
apk6	15	40	17
apk7	1	111	57
apk8	9	103	48
apk9	9	99	33
apk10	10	68	19
apk11	16	47	41
apk12	13	88	31
...	...	...	...
apk115	5	117	42
apk116	14	52	38
apk117	13	41	46
apk118	10	108	15
apk119	3	122	38
apk120	8	60	37
VariantDEX	11	110	43
VariantRES	4	48	59
VariantBOTH	20	93	18

of patterns in common with the one input so that Puppetdroid is able to find the same type of view during re-execution of input. We compared our approach with two opposite Android similarity approach, PHash and Androsim. We saw that *SimDroid-UI* was the best in terms of visual similarity produced because, using structural data as similarity measure, it can compare the hierarchy of the views instead of (1) coding (Androsim) and (2) final appearance (PHash). Finally we saw that in terms of execution, our clustering approach, as for PHash, allows to search a new sample in the whole dataset without looking every single data, and scaling well as dataset size increases.

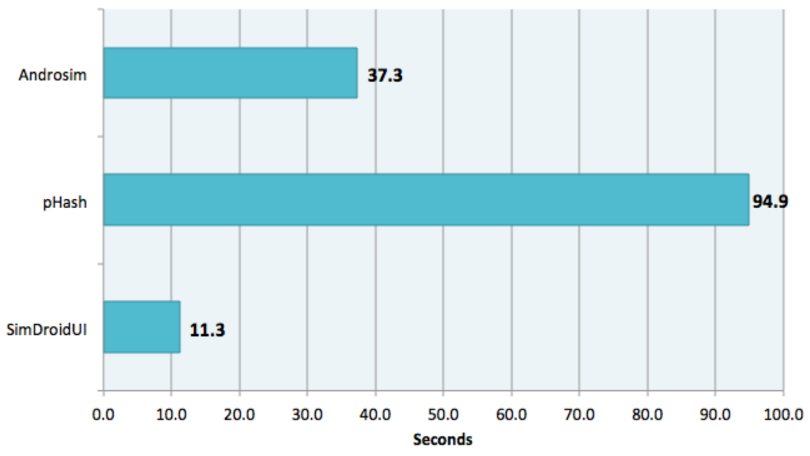


Figure 5.8: Execution time comparison of similarity computation

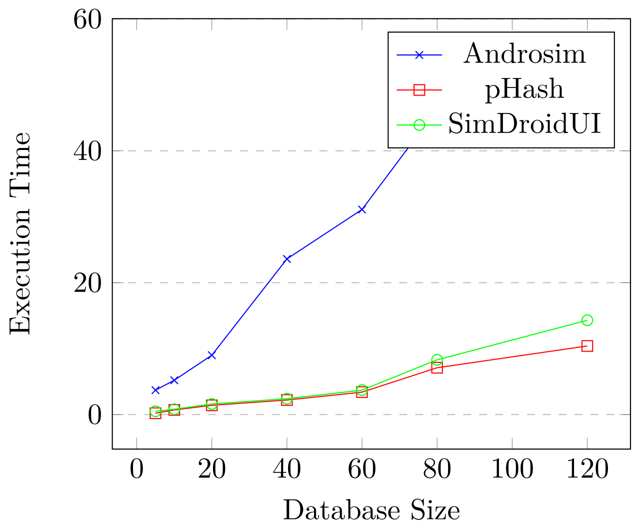


Figure 5.9: Execution time for search with respect to dataset size

Chapter 6

Conclusions and future work

The goal of our work was to propose a different similarity measure for Android applications in order to compare applications with respect to the UIs results instead of the source code. Our key intuition is that file that represents the UI elements can be represented as trees and for comparing UIs we can compare the structure of these trees. We designed and developed *SimDroid-UI* , a tool for grouping similar applications with respect to their visual appearance. We added a search functionality to quickly compare a new sample with clusters (groups), created in order to obtain a list of similar applications without seeking into the whole database.

In order to build *SimDroid-UI* , we dealt with various challenges. Firstly, we faced the problem of extracting a tree representation of an APK, by combining informations from several decompiled files: the problem was solved applying techniques to combine XML files, and using ElementTree Python module to convert the resulting XML files into a tree. Secondly, we faced the problem of retrieving common structures among the whole dataset; we decided to use CMTreeMiner (Section 3.1.4), a tool for mining frequent subtrees in a database of rooted unordered trees. Third, we analyzed the problem of creating groups of applications by comparing substructures: we defined a similarity measure using the Jaccard Similarity value computed between two sets and we applied a hierarchical clustering algorithm in order to obtain clusters of applications. We had experimentally evaluated a good stopping criterion for the algorithm and defined a representation of a cluster able to summarize the applications included.

Then we defined a way to compare a new sample to the created clusters. This brought to quickly associate a new sample to a group of applications. We believe the searching process can help in different situations both for security and recommendation process (extensively used in markets).

We tested our solution using Puppetdroid, a tool for automatic dynamic analysis of Android malwares, in order to get a real evaluation of the similarity application returned by *SimDroid-UI* .

Our experiments proved that clusters, created by *SimDroid-UI* , can be considered a good representation of the UI similarity among applications and that each group could be associated to an application type. Furthermore, we experimentally demonstrated that our searching process is able to return applications that have similar structural representation of the UI in less time with respect to pHash and Androsim approaches.

Our main contribution to the State of the Art have been (1) the definition of a new approach to compare applications based on UI and (2) the development of an efficient searching process to quickly retrieve similar applications that is not much influenced by the dataset size. Moreover, we created a request manager module to process application and retrieve similar applications transparently.

We plan to fully integrate *SimDroid-UI* in Puppetdroid, in order to improve its results as we demonstrated, by using our request manager.

Moreover, we plan to scale our product to use different clustering process and different Tree Mining algorithms as alternatives to CMTreeMiner. In order to achieve this, we have to allow the *SimDroid-UI* user to choiche which cluster algorithm and which Tree Mining algorithm use. This would be done by modifying request manager to become a full API module that can be transparently used by an user.

Appendix A

Bipartite Matching Problem

A.1 Maximum Bipartite Matching

A matching in a Bipartite Graph is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In other words, a matching is maximum if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph. There are many real world problems that can be formed as Bipartite Matching. For example, consider the following problem: there are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such that as many applicants as possible get jobs.

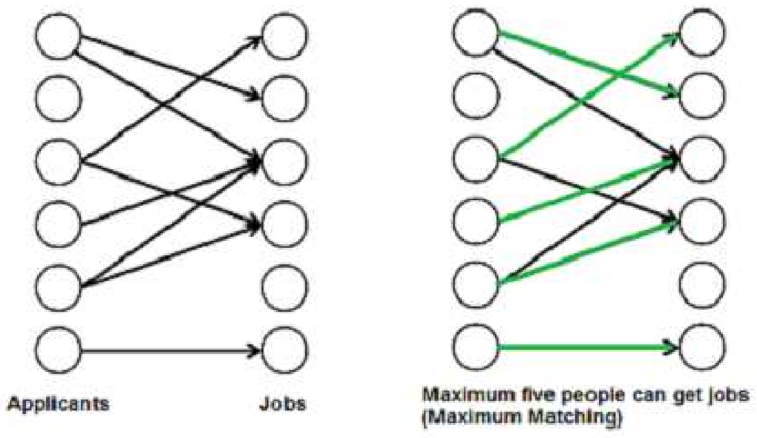


Figure A.1: Bipartite Matching example

A.2 Maximum Bipartite Matching and Max Flow Problem

To face the problem it needs to consider the Ford–Fulkerson method. This is an algorithm which computes the maximum flow in a flow network. It was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson. The idea behind the algorithm is as follows: as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path. So the Maximum Bipartite Matching (MBP) problem can be solved by converting it into a flow network. Following are the steps:

1) *Build a Flow Network.* There must be a source and sink in a flow network. So we add a source and add edges from source to all applicants. Similarly, add edges from all jobs to sink. The capacity of every edge is marked as 1 unit.

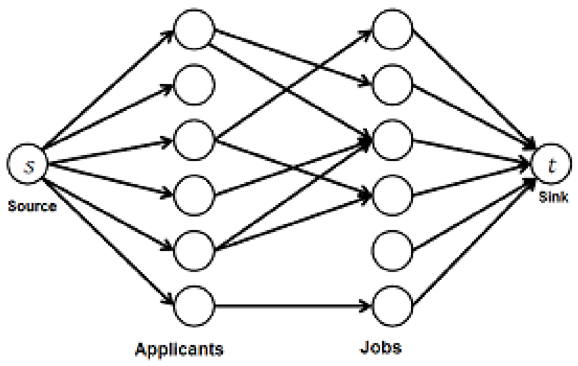


Figure A.2: An example of flow graph

2) *Find the maximum flow.* We use Ford-Fulkerson algorithm to find the maximum flow in the flow network built in step 1. The maximum flow is actually the MBP we are looking for.

To implement the above approach first of all there are to defineLet us first define input and output forms. Input is in the form of Edmonds matrix which is a 2D array ‘bpGraph[M][N]‘ with M rows (for M job applicants) and N columns (for N jobs). The value bpGraph[i][j] is 1 if i’th applicant is interested in j’th job, otherwise 0.
Output is number maximum number of people that can get jobs.

A simple way to implement this is to create a matrix that represents adjacency matrix representation of a directed graph with M+N+2 vertices. Call the fordFulkerson() for the matrix. This implementation requires O((M+N)*(M+N)) extra space. Extra space can be be reduced and code can be simplified using

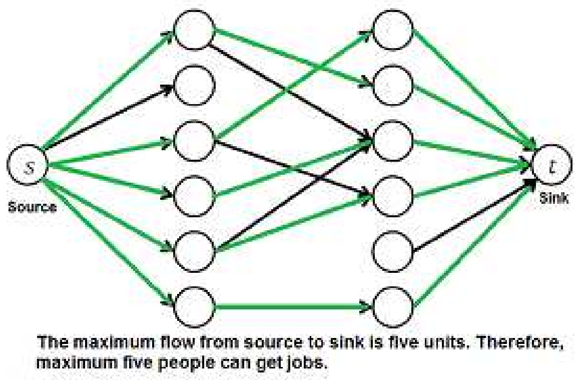

s
Source
t
Sink
The maximum flow from source to sink is five units. Therefore, maximum five people can get jobs.

Figure A.3: Maximum flow example

the fact that the graph is bipartite and capacity of every edge is either 0 or 1. The idea is to use DFS traversal to find a job for an applicant (similar to augmenting path in Ford-Fulkerson). We call bpm() for every applicant, bpm() is the DFS based function that tries all possibilities to assign a job to the applicant.

In bpm(), we one by one try all jobs that an applicant 'u' is interested in until we find a job, or all jobs are tried without luck. For every job we try, we do following.
If a job is not assigned to anybody, we simply assign it to the applicant and return true. If a job is assigned to somebody else say x, then we recursively check whether x can be assigned some other job. To make sure that x doesn't get the same job again, we mark the job 'v' as seen before we make recursive call for x. If x can get other job, we change the applicant for job 'v' and return true. We use an array maxR[0..N-1] that stores the applicants assigned to different jobs. If bmp() returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in maxBPM().

```
// A C++ program to find maximal Bipartite matching.
#include <iostream>
#include <string.h>
using namespace std;

// M is number of applicants and N is number of jobs
#define M 6
#define N 6

// A DFS based recursive function that returns true if a
// matching for vertex u is possible
bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR[])
{
// Try every job one by one
for (int v = 0; v < N; v++)
{
// If applicant u is interested in job v and v is
// not visited
if (bpGraph[u][v] && !seen[v])
{
seen[v] = true; // Mark v as visited

// If job 'v' is not assigned to an applicant OR
// previously assigned applicant for job v (which is matchR[v])
// has an alternate job available.
// Since v is marked as visited in the above line, matchR[v]
// in the following recursive call will not get job 'v' again
if (matchR[v] < 0 || bpm(bpGraph, matchR[v], seen, matchR))
{
matchR[v] = u;
return true;
}
}
}
return false;
}
```

Listing A.1: BPM C++ code

```
// Returns maximum number of matching from M to N
int maxBPM(bool bpGraph[M][N])
{
// An array to keep track of the applicants assigned to
// jobs. The value of matchR[i] is the applicant number
// assigned to job i, the value −1 indicates nobody is
// assigned.
int matchR[N];

// Initially all jobs are available
memset(matchR, −1, sizeof(matchR));

int result = 0; // Count of jobs assigned to applicants
for (int u = 0; u < M; u++)
{
// Mark all jobs as not seen for next applicant.
bool seen[N];
memset(seen, 0, sizeof(seen));

// Find if the applicant 'u' can get a job
if (bpm(bpGraph, u, seen, matchR))
result++;
}
return result;
}
```

Listing A.2: maxBPM C++ code

```
// Driver program to test above functions
int main()
{
// Let us create a bpGraph shown in the above example
bool bpGraph[M][N] = {
{0, 1, 1, 0, 0, 0},
{1, 0, 0, 1, 0, 0},
{0, 0, 1, 0, 0, 0},
{0, 0, 1, 1, 0, 0},
{0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 1}
};

cout << "Maximum number of applicants that can get job is "
<< maxBPM(bpGraph);

return 0;
}
```

Listing A.3: BPM driver program example

Output: Maximum number of applicants that can get job is 5

Bibliography

[1] H. Shewale, S. Patil, V. Deshmukh, and P. Singh, "Analysis of android vulnerabilities and modern exploitation tecniques," *Ictact Journal on Communication Technology*, vol. volume 05, pp. 863–867, March 2014.

[2] C. Maia, L. Nogueira, and P. L.M., "Evaluating android os for embedded real-time systems," *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010), Brussels, Belgium.*, no. TR 100604, pp. 63–70, 2010.

[3] "Google inc. android user interface." `http://developer.android.com/guide/topics/ui/index.html`.

[4] Wikipedia The Free Encyclopedia, "Android." `http://en.wikipedia.org/wiki/Android`.

[5] "Market share statistics for internet technologies - mobile/tablet operating system market share." `http://www.netmarketshare.com/operating-system-market-share.aspx`, Sept. 2014.

[6] IDC, "Smartphone os market share." `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, 2014.

[7] S. Holla and M. Katti, "Android based mobile application development and its security," *International Journal of Computer Trends and Technology, Department of Information Science & Engg, RV College of Engineering, Bangalore, India*, pp. 486–490, 2012.

[8] "Android." `https://source.android.com/devices/tech/index.html`.

[9] Open Signal, "Android fragmentation." `http://opensignal.com/reports/2014/android-fragmentation`, 2014.

[10] A. Menti and M. Zago, "Attacco ad android: obiettivi, rischi e contromisure." `http://profs.scienze.univr.it/~mastroen/Attacco_ad_Android_Menti_Zago.pdf`, 2013.

[11] Wikipedia The Free Encyclopedia, "Google play." `http://en.wikipedia.org/wiki/Google\_Play`.

[12] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis, "Andradar: Fast discovery of android applications in alternative markets," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (S. Dietrich, ed.), pp. 51–71, 2014. In Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). London, UK.

[13] Sophos Mobile, "Mobile security threat report 2014," 2014. In Proceedings of 2014 Mobile World Congress by Vanja Svajcer, Principal Researcher, SophosLabs.

[14] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," *NDSS*, Feb. 2012.

[15] Y. Joung Ham, D. Moon, H. Lee, J. Deok Lim, and J. Nyeo Kim, "Android mobile application system call event pattern analysis for determination of malicious attack," *International Journal of Security and Its Applications*, vol. 8, no. 1, pp. 231–246, 2014.

[16] L. M. Security, "Lookout mobile security: State of mobile security 2012." Technical report, Lookout Mobile Security, September 2012.

[17] C. Crussell, J. and Gibler and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Computer Security – ESORICS 2012* (S. Foresti, M. Yung, and F. Martinelli, eds.), pp. 37–54, 2012. In Proceedings of the 17th European Symposium on Research in Computer Security, Pisa, Italy.

[18] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *36th International Conference on Software Engineering(ICSE 2014)* (A. York, ed.), pp. 175–186, 2014. In the Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, May 31-June 07, 2014.

[19] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys*, Feb 2012.

[20] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *ACSAC'07*, pp. 421–430, 2007. In Proceedings of 23rd Annual Computer Security Applications Conference, Miami Beach.

[21] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro, and S. Zanero, "Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications," *ArXiv e-prints*, Feb. 2014. Computer Science - Cryptography and Security.

[22] S. Hanna, L. Huang, S. Wu, E. and Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (U. Flegel, E. Markatos, and W. Robertson, eds.), pp. 62–81, July 2012. In Proceedings of 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece.

[23] R. Duda, P. Hart, and D. Stork, *Pattern Classification.* John Wiley and Sons, 2000.

[24] S. Li, "Juxtapp and dstruct: Detection of similarity among android applications," Master's thesis, EECS Department, University of California, Berkeley, May 2012.

[25] M. Zaki, "Efficiently mining frequent trees in a forest," ACM New York, July 2002. In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

[26] Y. Chi, Y. Yang, Y. Xia, and R. Muntz, "Cmtreeminer: Mining both closed and maximal frequent subtrees," in *Advances in Knowledge Discovery and Data Mining*, Springer Berlin Heidelberg, May 2004. In Proceedings of 8th Pacific-Asia Conference, PAKDD 2004, Sydney, Australia, May 26-28, 2004.

[27] K. Wang and H. Liu, "Discovering typical structures of documents: A road map approach," ACM New York, Aug. 1998. In Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.

[28] Y. Xiao, J. Yao, Z. Li, and M. Dunham, "Efficient data mining for maximal frequent subtrees," IEEE Computer Society Washington, Nov. 2003. In Proceedings of the Third IEEE International Conference on Data Mining.

[29] Y. Chi, Y. Xia, Y. Yang, and R. Muntz, "Mining closed and maximal frequent subtrees from databases of labeled rooted trees," *IEEE*, 2005.

[30] M. Lee, L. Yang, W. Hsu, and X. Yang, "Xclust: Clustering xml schemas for effective integration," in *In Proceedings of the Eleventh International Conference on Information and Knowledge Management*, pp. 292–299, 2002.

[31] *Xproj: A Framework for Projected Structural Clustering of XML Documents*, 2007.

[32] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. Sellis, "Clustering xml documents using structural summaries," 2004. In Proceedings of EDBT 2004 Workshops, Heraklion, Greece, March 14-18 2004.

[33] N. Mamoulis, W. Cheung, and W. Lian, "Similarity search in sets and categorical data using the signature tree," pp. 75–86, 2003. In Proceedings of International Conference On Data Engineering, 2003.

[34] A. Jain and R. Dubes, *Algorithms for Clustering Data.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1998.

[35] S. Chawathe, S. Abiteboul, and J. Widom, "Managing historical semistructured data," *Theory and Practice of Object Systems*, vol. 5, pp. 143–162, 1999.

[36] W. Lian, D. Cheung, N. Mamoulis, and S. Yiu, *An Efficient and Scalable Algorithm for Clustering XML Documents by Structure.* IEEE, NJ, USA, 2004.

[37] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. Sellis, "A methodology for clustering xml documents by structure," *Informastion Systems*, vol. 31, pp. 187–228, 2006.

[38] F. Luccio, A. Enriquez, P. Rieumont, and L. Pagli, "Exact rooted subtree matching in sublinear time," in *In Proceed. 9-th ANaCC/ACM/IEEE Intern.Congress on Comp.Sc. (CIIC'02)*, pp. 27–35, Accademic Pubblication Ltd, 2002.

[39] F. Luccio, A. Enriquez, P. Rieumont, and L. Pagli, "Bottom-up subtree isomorphism for unordered labeled trees," in *International Journal of Pure and Applied Mathematics*, vol. 38, pp. 325–343, Accademic Publication Ltd, march 2007.

[40] "Androguard." `http://code.google.com/p/androguard`.

[41] P. Refaeilzadeh, L. Tang, and H. Liu, "Cross-validation," in *Encyclopedia of Database Systems*, pp. 532–538, Springer US, 2009.

Acronyms

AOSP *Android Open Source Project*
API *Application Programming Interface*
BER *Bit Error Rate*
CFG *Control Flow Graph*
DCT *Discrete Cosine Transform*
DEX *Dalvik EXecutable*
DM *Data Mining*
DTD *Document Type Definition*
DVM *Dalvik Virtual Machine*
HTML *HyperText Markup Language*
JAR *Java ARchive*
JVM *Java Virtual Machine*
KDD *Knowledge Discovery in Databases*
NCD *Normalized Compressed Distance*
PCC *Peak of Cross Correlation*
URL *Uniform Resource Locator*
VM *Virtual Machine*
VNC *Virtual Network Computing*
XAML *eXtensible Application Markup Language*