SIMILARITY DETECTION TECHNIQUES FOR MOBILE PLATFORM ARTIFACTS

BY AMRUTA GOKHALE

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Dr. Vinod Ganapathy

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2015

ABSTRACT OF THE DISSERTATION

Similarity Detection Techniques for Mobile Platform Artifacts

by Amruta Gokhale

Dissertation Director: Dr. Vinod Ganapathy

There has been a tremendous increase in availability and use of mobile apps in recent years. Two dominant mobile platform providers, Apple and Google, have over 1.4 million applications (apps) each in their app markets as of May 2015. Such huge number of available apps gives rise to problems in managing app repositories. Also, a peculiar set of challenges is faced by the large community of mobile app developers.

This dissertation describes novel solutions to two problems in the space of mobile apps. The first problem is that of fragmentation of mobile app development across multiple platforms. To accelerate the native development of same app across multiple platforms, mobile app developers need tools to better navigate across different platforms. The second problem is that of keeping app repositories free of plagiarized apps. It is in the best interest of users, developers as well as app repository owners to get rid of plagiarized apps in the app markets, which often carry malicious software with them, and thus pose a threat.

Every mobile app interacts with the platform, taking advantage of the functionalities exposed by the platform. The behavior of an app can be described at a high level in terms of its interaction with the underlying platform. By monitoring the app and collecting such interaction in the form of API method invocations, we can record its high level behavior. We use this observation to build two new systems. The first system provides assistance in cross-platform mobile app development. The second system detects plagiarized mobile apps in app repositories. The

proposed techniques intercept the interactions between mobile apps and the underlying platform and utilize the interactions to infer likely resemblances between two mobile platform APIs or between mobile apps.

Acknowledgements

As I embarked on this journey of procuring the doctorate degree, there were many people who played roles in various capacities which helped me get to the final destination. I express my heartfelt gratitude towards them.

My advisor, Dr. Vinod Ganapathy, has helped shape many ideas behind the research projects I worked on. His excitement about solving new problems was infectious. His insights about various possible solutions to problems proved very helpful. I would like to thank him for being an excellent research advisor. I would also like to thank Dr. Liviu Iftode for helping me understand the big-picture view of the research problems I was working on. His comments on presentation slides helped improve my conference talks.

I would like to thank my committee members, Dr. Alex Borgida, Dr. Santosh Nagarkatte and Dr. Pratyusa Manadhata, for providing me feedback to improve this dissertation. I would also like to thank my collaborators, Dr. Ulrich Kremer, Daeyoung Kim, Yogesh Padmanaban and John McCabe from Rutgers and Abhinav Srivastava from AT&T, for their contributions to the joint research projects.

I had the fortune of making some wonderful friends during my graduate school period. Chathra Hendahewa, Priya Govindan and Rezwana Karim formed the group with whom I could share the high's and low's of graduate school. I would like to thank past and present members of disco-lab, specially Arati Baliga, Lu Han, Liu Yang, Mohan Dhawan, Shakeel Butt, Rezwana Karim, Daeyoung Kim, Hai Nguyen and Nader Boushehrinejadmoradi. They provided valuable feedback on my presentations and talks. I had interesting technical as well as non-technical discussions with them.

I owe a debt of gratitude to my parents – aai and baba. When I was growing up as a kid, my parents sowed the seeds in me to perform to the best of my ability in anything I do. They provided me with the best possible resources, for which they often had to sacrifice some part

of their own lives. For aai, my mother, the first priority was always her kids; everything else was secondary. I never can pay back the debt I owe to her. I have happy and fond memories of spending my younger years with my sister, Ruta, and my brother, Chaitanya. Conversations with them as well as my cheerful niece, Maitreyee, is something I always look forward to. Their love and support gives me the needed strength. Last, but certainly not the least, I attribute my success to my husband – Ajay – who stood behind me like a powerful force when needed. His encouragement, patience and understanding over all the years was instrumental in driving me from the beginning of this journey till the very end.

Dedication

To, Aai & Baba, for the unwavering support and unconditional love

To, Ajay, for the true companionship

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation describes novel solutions to two similarity detection problems in the space of mobile apps. The first problem is inferring similarities between API methods of two mobile platforms. The second problem is finding similarities between mobile apps of a single platform with application to detecting plagiarism in them.

1.1 Motivation

In the last few years, we have been witnessing the gradual decline of desktop computers and the rapid rise of mobile devices for accessing a majority of digital content. According to the recent statistics [24], mobile traffic to websites will overtake desktop traffic by March 2017. Fueling this growth in the adoption of mobile devices is the huge number of mobile apps being made available on mobile platforms. The two most popular mobile platforms, iOS and Android, have more than one million apps each in their app markets.

If you compare the landscape of development of mobile apps with that of desktop software, you will find many differences. The differences are visible in categories such as the affiliation and skill set of developers, the size of developers' community, the kind of repositories to upload software, and the functionalities provided by the platform. To give an example, Apple announced in June 2015 [14] that the iOS app store had 1.5 million apps available, with 100 billion number of downloads of apps from the app store and that $30 billion has been paid out to developers to date. As per the data released by app figures in 2014 [4], Google Play has over 350,000 registered developers where as iOS App store attracted over 250,000 developers. This is at an unprecedented scale, in terms of the size of developers' community, the number of apps available at the app market and the total number of downloads of the software. Let us look at these differences in greater details.

Until a few years ago, the task of developing applications was largely confined to teams of software engineers, either in the open-source community or at IT companies. Most of these engineers were professional programmers employed by IT firms, with an exception of small number of developers of open source software. Almost all of the commercial software development was done in-house. For example, the software applications for Microsoft's Windows operating system were written by employees and contractors employed by Microsoft and could never be done by third-party developers.

In contrast, the development of mobile apps is usually driven by small teams consisting of one or two developers, as noted in [82]. Many of these developers are either individual contributors or employees of small IT firms. These developers write apps for a platform which is owned and released by a company different than the one they are affiliated with. For example, it is not uncommon for individual contributors or programmers employed by small IT firms to write apps for Microsoft's Windows Phone platform, even though their parent IT firm does not own the platform. Such teams (or individuals) may lack the expertise and experience of a large team of in-house developers. Mobile app developers therefore tend to rely heavily on the functionality provided by the underlying mobile platform [82]. It is imperative for the mobile platforms to come equipped with a rich set of APIs so as to make the development easier for the third-party app developers.

The concept of app stores was non-existent until the arrival of mobile devices. The company that would make commercial software would distribute it via it's own distribution channel. Since the software was being developed in-house, and was usually closed to third-party developers, the company would have complete control over the distribution channel. As a result, managing the distribution channel or the software repository owned by the company was relatively easy. For example, although the issue of plagiarism of desktop software was rampant, such plagiarized software could never gain an entry on the software repository owned by the maker of original software. The in-house desktop software development also limited the scale at which software applications were uploaded and distributed. This made the management of software repositories easier. In case of mobile platforms, the humongous number of apps being deployed to the app stores gave birth to various issues associated with the management of large software repositories. Among these issues, we will look at detecting plagiarized mobile apps.

This dissertation is divided into two parts. In the first part, we are focusing our attention on mobile app development and the challenges faced by the developers of mobile apps. In the second part, we will look at how to detect plagiarized mobile apps.

One challenge in mobile app development is the fragmentation of app development process across multiple mobile platforms. Development on different mobile platforms differs on many fronts such as use of different programming languages, support for specific development environments and different Software Development Kits (SDKs). Fragmentation also exists within the same platform in the form of different screen sizes, various screen resolutions and different flavors or versions of the same operating system. This requires maintaining different code bases, thus leading to increased maintenance overhead. Mobile app developers also need to have better support for monitoring, analysis and testing of mobile apps. Joorabchi *et al.* conducted an exploratory study [51] to understand the current practices and difficulties in native mobile app development. Among all the different challenges expressed by developers, 76% of survey participants expressed concerns over app development for multiple mobile platforms. As per the survey, *"One type of challenge mentioned by many developers is learning more languages and APIs for the various platforms"*. We have attempted to address this challenge by developing a tool that would help developers to familiarize themselves with the API of a newer platform.

Every mobile platform is shipped with APIs which the programmers can use to perform common tasks in the implementation of the business logic of the app. For example, in Android, `drawRect()` can be used to draw a rectangle on the screen. `setColor()` can be invoked to set the color of the palette. These methods provide abstractions for common tasks, thus freeing the programmers from writing low level code. Programmers therefore find it easy and convenient to use the platform libraries to build apps on top of the respective platform. Hence, it is reasonable to expect that almost all mobile apps will interact with the underlying platform during their execution in the form of invocations to the library methods.

The interaction of the app with the underlying platform serves as a useful resource. We can observe this interaction and use it as an abstract representation for the behavior of the app. We use this interaction as an asset to solve two problems in the space of mobile apps. The first problem is how to automatically infer mappings between API methods of two platforms.

The second problem is how to detect plagiarized mobile apps that have been produced by fraudulently copying other apps. Next, we describe the two problems in more detail and the different techniques that we have employed to generate and use the interaction of the app with the platform.

1.2 Detecting Similarities or Mappings between Mobile Platform APIs

Today's mobile market is dominated by three major platforms, namely iOS, Android and Windows Phone. Developers often wish to make their apps available across as many of these platforms as possible so as to increase their user base. These developers face the challenge of developing apps separately for each new platform. Small teams of developers cannot afford to allocate separate resources to develop apps for each individual platform. Thus, we need tools to help such developers who are looking to develop apps across multiple platforms.

One solution for app development across multiple platforms is to develop cross-platform mobile apps. There are a number of ways to achieve the goal of developing cross-platform mobile apps. One approach is to develop apps using portable frameworks, such as HTML5 or JavaScript, which are supported by most mobile platforms. These apps can be augmented with hybrid frameworks (e.g., PhoneGap [19], Sencha [21], and MobileFirst [16]) to allow access to local resources on the mobile devices. While the resulting apps are portable, their reliance on the underlying machinery to parse and render HTML5 and JavaScript, impacts their runtime performance. Thus, app developers eager to ensure low performance overheads often prefer native app development, in which apps are developed using each platforms API. In this dissertation, we are focusing our attention only on solutions to develop native apps.

Porting apps from one platform to another is one method that makes apps available across platforms, yet allows to develop apps natively on each platform. However, similar to the task of porting desktop software to different desktop platforms, porting a mobile app across different mobile platforms is challenging. To illustrate, let us assume that we want to port an iPhone app to Android platform. Various aspects of the app that need to be changed are: (1) Programming language (Objective-C vs. Java) (2) Structure of source code (Model-View-Controller design in iOS vs four components in Android consisting of Activities, Services, Content Providers and

Broadcast Receivers) (3) Platform-specific APIs (iOS API vs Android API), and (4) Platform-specific customization. Among these, let us focus our attention on platform-specific APIs. We must modify the app to use Android API (*target* API) in place of iOS API (*source* API). To accomplish this task, we need to identify Android API methods that perform the same function as iOS API methods.

However, identifying functionally equivalent methods between two APIs is difficult. It can be done manually by consulting the documentations of both the APIs. But the manual process is time consuming and error-prone. One solution to this problem is to automatically build a *database* of mappings between the source and target APIs. The database contains mappings consisting of each source API method paired with a target API method implementing the same functionality. It is possible to have each source API method mapped to multiple target API methods, representing various different ways in which the same functionality can be implemented on the target platform. Developers who wish to port their iPhone apps to Android would lookup this database to find equivalent Android methods that can replace existing iOS methods in the source code of the app.

The goal thus is to *automate* the process of inference of mappings between APIs of two mobile platforms. What is the right form of input that would enable us to find API mappings? Previous work [95] that mined API mappings between APIs of two languages used aligned source code of same projects that were available in both the languages as input. They assumed the existence of same projects being available in both the languages, where one of the projects was most likely produced by translation from the other. We chose to use actual API usages in the app as input. These API usages are nothing but invocations to API methods made by the app to achieve the desired functionalities. We describe two approaches to solve the problem of automatically inferring API mappings. At a high level, both the techniques use the following approach:

1. Generate API usages for each set of apps on the two platforms
2. Use an inference technique to establish likely mappings between individual API elements in the API usages

The two approaches differ in the methods they follow to generate the API usages, including

the kind of apps they use as input and also in the techniques employed to infer the mappings. The first approach involves dynamically executing the apps and relies on availability of apps on two platforms that offer the same functionality to the end user. The second approach works statically and does not have this requirement.

1.2.1 Dynamic Approach to Infer Mappings

We develop **Rosetta** (Chapter 2), a novel approach to infer mappings between APIs of a source and a target platform. It is built on the assumption that there exist pairs of independently developed applications on both the platforms which offer similar functionality at a high level. For example, two TicTacToe games, both of which may have been developed independently on two mobile platforms, would constitute one such pair. The key observation we make is that the developers of these applications exercised knowledge of the corresponding APIs while building these applications. We develop a technique to systematically harvest this knowledge and infer likely mappings between the APIs of the source and target platform.

We take a pair of apps offering same functionality on both platforms, execute them by providing the same input, and collect the execution traces. The traces contain API method invocations made by the apps while interacting with the respective platforms. The idea is to collect multiple such trace pairs from sets of apps on the two platforms, build a graphical model of possible API mappings and their likelihoods, and use a probabilistic inference algorithm to find mappings that are most likely to be true.

1.2.2 Static Approach to Infer Mappings

We propose another approach **DDR** (Section §2.6), that infers mappings between APIs of a source and a target platform by static analysis of apps. This approach is inspired by a technique in natural language processing (NLP) domain which extracts a translation dictionary from non-parallel corpora of two natural languages. The NLP technique infers mappings between words of two languages by taking sentences of two languages as input. We apply the technique to the domain of mobile apps, treating software code in the same light as NLP text. We first create program paths containing API method usages, constructed by static analysis of the apps. These program paths become *sentences* of some hypothetical language. The *words* in this

language are individual API methods. A collection of program paths becomes the text corpus of the language. We feed two such collections to the NLP tool to infer mappings between API methods.

1.3 Detecting Similarities between Mobile Apps

Many industries have been plagued by plagiarized work, including the world of books, art and music. In software industry, code plagiarism has been a known problem and several solutions have been proposed ([27, 31, 52, 56]) to detect it. Code plagiarism is the act of copying software code written by others without obtaining their permission. Mobile software industry is no exception to the existence of plagiarized artifacts. With an exponential growth in the number of mobile apps being made available in popular app markets, plagiarized mobile apps are plentiful in the app markets as shown by previous studies ([99, 98]). A plagiarized mobile app is an app that was copied from another app without obtaining the necessary permission from the developer of the original app.

Plagiarized apps present in the app markets are a nuisance to users and may often lead to unwanted consequences such as data breach or loss of private information. Since plagiarized apps look and feel exactly the same as original apps, and could be offered for free as opposed to the paid original apps, users are lured into downloading them. From the perspective of developers of original apps, plagiarized apps may result in loss of potential user base and possibly revenue loss [43]. Moreover, plagiarized apps have recently been used as a carrier for infecting the mobile devices with malware [99]. Developers of such malicious apps first obtain a copy of the original app, then modify the reverse engineered app to inject the malware, and then upload this repackaged app as a new app on the app market. Unsuspecting users download the plagiarized app and become victims of this attack.

It is therefore important to detect plagiarized apps in the app markets. Identifying such apps requires us to create a unique representation of apps which is short enough for comparison yet carry sufficient description of high level behavior of the app. Following two conditions are desirable for a unique representation of an app that can be used to detect plagiarism:

- If two apps are not plagiarized, their unique representations should be different. Likewise, if two apps are plagiarized, their representations should be similar.
- It should be feasible to measure similarity between the chosen representations of two apps by some quantitative measure.

One way to detect plagiarized apps is to deploy anti-malware tools that would detect the possibly malicious nature of such apps. Mobile anti-malware tools work in the same way as that for desktops. The most common technique used by these tools is signature matching in which signature of the given app is compared against those of known malware in the signature database. However, such signature-based malware detection can be easily defeated by using simple transformations [33]. Such transformations include variable and function renaming, and code reordering, which alter the syntactic structure of the code, so that it no longer matches the known signatures.

In response, the community has proposed many techniques that detect plagiarized mobile apps even in the presence of transformations ([47], [98] and [97]). Most of these techniques statically analyze the app to compute syntactic features such as unique code hash [98] or the set of permissions requested by the app [97]. We show that such syntactic similarity detection techniques can be defeated by building Androcrypt (Section §3.2), an obfuscator that uses simple code encryption to produce obfuscated apps. Inspired by prior work [68] to create unique fingerprints of desktop programs, we develop a robust, dynamic approach to detect plagiarized mobile apps (Section §3.3). The key idea is that an app can affect the state of the mobile device only by interacting with the platform. Thus, similar apps must interact with the platform in similar ways. We capture this interaction and build what is called as *API birthmark* of the app. We use this representation for similarity detection.

1.4 Summary of Contributions

The thesis this dissertation supports is:

The interactions of mobile apps with the underlying platform can be leveraged to infer likely resemblances among platform APIs or apps.

This dissertation supports the above thesis statement and makes the following contributions:

- We present two approaches to automatically infer mappings between API methods of two mobile platforms. Rosetta (Chapter 2) infers likely mappings between APIs of two mobile platforms by using a few independently developed applications on the two platforms that implement same functionality. As output, it produces a ranked list of mappings inferred for each source API method seen in the input trace. DDR - Data Driven Rosetta - (Section §2.6) is a tool to infer mappings between two APIs by using a technique in NLP domain that has been successfully applied to infer mappings between words of two natural languages. DDR uses this technique in the domain of mobile apps to infer API mappings.

- We demonstrate our approaches to infer API mappings automatically by building two prototype tools. We design and implement Rosetta, a tool that infers mappings between JavaME and Android graphics APIs. We implement DDR, a tool to infer mappings between iOS and Android API.

- We show that static similarity detection techniques which use syntactic features of the app as the basis for comparison can be defeated by use of simple code encryption. We show the failure of these tools to detect similarity between an original app and the same app transformed by Androcrypt, a proof-of-concept obfuscator that uses code encryption.

- We propose a dynamic technique to detect similarity based on constructing *API birthmarks* which are nothing but unique fingerprints of the app constructed by intercepting the runtime interaction between the app and its underlying platform.

1.5 Statement of Contributions

The following is a list of people who co-authored papers from which material was used in this dissertation. My dissertation advisor Professor Vinod Ganapathy was involved in the design,

implementation and evaluation of all three projects. Yogesh Padmanaban was involved in the implementation of Rosetta. Specifically, he collected traces of a subset of Android apps in our dataset. Daeyoung Kim from Rutgers and Abhinav Srivastav from AT&T Labs were my collaborators in the API birthmark project. Daeyoung Kim implemented Androcrypt obfuscator, collected traces of Android apps using the monkey tool and contributed in the experimentation and evaluation. Daeyoung Kim was also involved in the DDR project. He implemented the trace collection module for iOS apps.

Chapter 2

Detecting Mappings between Mobile Platform APIs

We present an approach to detect similarities between APIs of two mobile platforms in this chapter. The similarities are nothing but the mappings between methods or method sequences of two mobile platform APIs that result in similar output when provided with the same input. The idea is to build a database of inferred API mappings. This database will serve as a useful resource in cross-platform app development to mobile app developers who are proficient on one platform but ignorant on the other. Such developers can lookup this database of mappings to retrieve the API methods of the unknown platform. We present a prototype tool called Rosetta that infers mappings between Java Platform, Micro Edition graphics API and Android graphics API.

2.1 Motivation

Software developers often wish to make their applications available on a wide variety of computing platforms. The developer of a gaming app, for instance, may wish to make his app available on smart phones and tablets manufactured by various vendors, on desktops, and via the cloud. The key hurdle that he faces in doing so is to *port* his app to these software and hardware platforms.

Why is porting software a difficult problem? Consider an example: suppose that we wish to port a Java Platform Micro Edition (Java ME)-based game to an Android-powered device. Among other tasks, we must modify the game to use Android's API [45] (the *target* platform) instead of Java ME's API [72] (the *source* platform). Unfortunately, the process of identifying the API methods in the target platform that implement the same functionality corresponding to that of a source platform API method is cumbersome. We must manually examine the SDKs of the source and target APIs to determine the right method (or sequence of methods) to use.

To add complexity, there could be multiple ways in which a source API method can be implemented using the target's API methods. For example, the `fillRect()` method in Java ME's graphics API, which fills a specified rectangle with color, can be implemented using either one of these two sequences of methods in Android's graphics API: `setStyle();drawRect()` or as `moveTo();lineTo(); lineTo();lineTo();lineTo();drawPath()` (we have omitted class names and the parameters to these method calls).

One way to address this problem is to populate a database of *mappings* between the APIs of the source and target platforms. In this database, each source API method (or method sequence) is mapped to a target API method (or method sequence) that implements its functionality. The database could contain multiple mappings (possibly ranked) for each source API method in cases where its functionality can be implemented in different ways by the target API. The mapping database significantly eases our task. We need only consider the mappings in this database to find suitable target API methods to replace a source API method, instead of painstakingly poring over the SDKs and their documentation. Such mapping databases do exist, but only for a few source/target API pairs (*e.g.,* Android, iOS and Symbian Qt to Windows 7 [87] and iOS to Qt [76]), and they are populated by domain experts well-versed in the source and target APIs.

Our contribution. We present an approach to automate the creation of mapping databases for any given source/target API pair. To bootstrap our approach, we rely on the availability of a few *similar application pairs* on the source and target platform. A source platform application $\mathcal{S}$ and a target platform application $\mathcal{T}$, possibly developed independently by different sets of programmers, constitute a similar application pair if they implement the same high-level functionality. For example, both $\mathcal{S}$ and $\mathcal{T}$ could implement the TicTacToe game on Java ME and Android, respectively. This situation is not uncommon in modern app markets, where independently-developed versions of popular apps are available in markets hosted by different vendors. Our approach builds upon the observation that *in implementing $\mathcal{S}$ and $\mathcal{T}$, their developers exercised knowledge about the APIs of the corresponding platforms*. We provide a systematic way to harvest this knowledge into a mapping database, which can then benefit other developers porting applications from the source to the target platform. Our approach works by

recording traces of $\mathcal{S}$ and $\mathcal{T}$ executing similar tasks, structurally analyzing these traces, and extracting likely mappings using probabilistic inference. Each mapping output by our approach is associated with a probability, which indicates the likelihood of the mapping being true. The intuition is that the more evidence we see of a mapping, *e.g.,* the same pair of API methods being used across many traces to implement the same functionality, the higher the likelihood of the mapping. The output of our approach is a ranked list of mappings inferred for each source API method.

We demonstrate our approach by building a prototype tool called ***Rosetta*** (in reference to the legendary Rosetta Stone) to infer likely mappings between the Java ME and Android graphics APIs. We chose Java ME and Android because both platforms use the same language for application development. However, this requirement is not germane to our approach, and it may be possible to adapt Rosetta to work with source and target APIs that use different languages for application development. We evaluated Rosetta with a set of twenty-one independently-developed Java ME/Android application pairs. Rosetta was able to find at least one valid mapping within the top ten ranked results for 71% of the Java ME API methods observed in our traces. Further, for 42% of Java ME API methods, the top-ranked result was a valid mapping.

2.2 Approach Overview

We present the workflow of our approach (Figure 2.1), tailored to Java ME and Android as the source and target platforms, respectively. We only provide informal intuitions here, and defer the details to §2.4. The workflow has four steps.

STEP 1: Collection of application pairs. The first step is to gather a database of applications in both the source and target platform. For each source application in the database, we require a target application that implements the same high-level functionality. For example, if we have a TicTacToe game for Java ME, we should locate a TicTacToe game for Android that is as functionally and visually (GUI-wise) similar to the Java ME game as possible.

Given the popularity of modern mobile platforms, and the desire of end-users to use similar applications across platforms, such application pairs are relatively easy to come by. Of course, given that the games were independently developed for these two platforms, there may

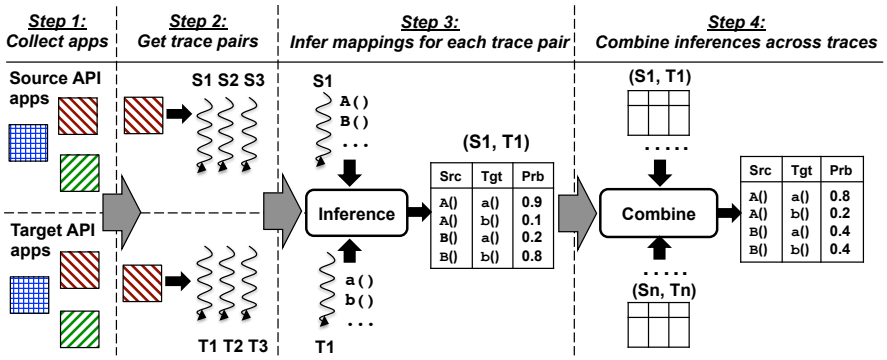

Figure 2.1: Workflow of our approach to inferring likely API mappings.

be minor differences in functionality. For example, the Android game may offer menu options that are different from those of the Java ME game. However, as we discuss in Step 2, we can restrict ourselves to inducing functionally similar execution paths in these applications. Any functional differences that still make their way into these execution paths will manifest as inaccuracies during probabilistic inference in Step 3. However, the effect of these inaccuracies can be mitigated by combining inferences across multiple applications pairs and their executions in Step 4.

STEP 2: Execution and collection of trace pairs. In this step, we take each application pair, and execute them in similar ways, *i.e.,* we provide inputs to exercise similar functionality in these applications. As we do so, we also log a trace of API calls invoked by the applications. This gives a *trace pair*, consisting of one trace each for the source and target applications. Figure 2.2 presents a snippet from a trace pair that we gathered for TicTacToe games on the Java ME and Android platforms. They were collected by starting the game, waiting for the screen to display a grid of squares, and exiting.

Since the traces in each pair were obtained by exercising similar functionality in the source and target applications, these traces must contain some API calls that can be mapped to each other. This is the key intuition underlying our approach. Of course, an application can be invoked in many ways, and in this step, we collect several trace pairs for each application pair. The output of this step is a database of functionally equivalent trace pairs ($\mathsf{Trace}_{\mathcal{S}}$, $\mathsf{Trace}_{\mathcal{T}}$) across all of the application pairs collected in Step 1.

STEP 3: Trace analysis and inference. In this step we analyze each trace pair to infer likely API mappings implied by the pair. Our inference algorithm relies on various attributes that

Java ME Trace	Android Trace
Graphics.setColor()	Paint.setStyle()
Graphics.fillRect()	Color.parseColor()
Graphics.setColor()	Paint.setStyle()
Graphics.fillRect()	Color.parseColor()
Graphics.fillRect()	Canvas.drawLine()
Graphics.fillRect()	Canvas.drawLine()
	Canvas.drawLine()
	Canvas.drawLine()
	Canvas.drawLine()

Note: Each trace snippet shows the method invoked, and the class in which the method is implemented. Java ME and Android classes are prefixed with `javax/microedition/lcdui` and `android/graphics`, respectively. For brevity, we only refer to method names and not their classes.

Figure 2.2: Snippets from traces of similar executions of Java ME and Android-based TicTacToe games.

.

are determined by the structure of the traces. We cast the attributes as inputs to a probabilistic inference algorithm (§2.4). The output of this step is a probability for each pair of source and target API calls `A()`/`a()` that indicates the likelihood of `A()` mapping to `a()` (denoted here as `A()`→`a()`). Our algorithm can also infer mappings between method sequences (*e.g.,* `A()`→`a();b()` and `A();B()`→`a()`).

We now discuss the attributes used by our approach using our running example. Our aim is to find likely mappings for the Java ME calls `setColor()` and `fillRect()`. For simplicity, we restrict our discussion to the snippets of the traces shown in Figure 2.2. In reality, our analysis considers the entire trace.

(**1**) *Call frequency.* If `A()` in the source API maps to `a()` in the target API, then the frequency with which these method calls occur in functionally-similar trace pairs must also match. The trace pairs may differ in the absolute number of method calls that they contain, so we focus on the *relative count* of each method call, which is the raw count of the number of times that a method is called, normalized by trace length. The table below shows the raw and relative counts of various method calls based upon the snippets in Figure 2.2. Using this attribute, the following API mappings appear likely, `setColor()`→`parseColor()`, `setColor()`→`setStyle()`, `fillRect()`→`drawLine()`, while the others appear unlikely. In fact, our approach works on method sequences as well using the same reasoning as above,

and infers that `setColor()→setStyle(); parseColor()` is a mapping.

API Call	Raw Count	Relative Count
`setColor()`	2	0.33
`fillRect()`	4	0.67
`setStyle()`	2	0.22
`parseColor()`	2	0.22
`drawLine()`	5	0.56

(**2**) *Call position.* The location of method calls in a trace pair provides further information to determine likely mappings. Since the traces exercise the same application functionality, method calls that map to each other should appear at "similar" positions in the trace pair. The table below shows the position of each of the method calls in our running example, roughly categorized as belonging to the first half or the second half of the corresponding traces. We can therefore reinforce the beliefs about API mappings inferred using call frequencies.

API Call	First Half	Second Half
`setColor()`	✓	×
`fillRect()`	×	✓
`setStyle()`	✓	×
`parseColor()`	✓	×
`drawLine()`	×	✓

(**3**) *Call context.* The context in which a method call `A()` appears is defined to be the set of API methods that appears in its vicinity in the execution trace (e.g., within a pre-set threshold distance, preceding or following `A()` in the trace). For example, both `setColor()` calls appear in the preceding context of `fillRect()` calls in the Java ME trace (using a threshold distance of 2 calls). Likewise, `parseColor()` and `setStyle()` appear in the preceding context of `drawLine()` in the Android trace. In fact, the sequence `setStyle(); parseColor()` appears in the preceding context of the first `drawLine()`. From this, we can infer that *if* `fillRect()→drawLine()` holds, *then* the mapping `setColor()→setStyle();parseColor()` is likely to hold.

(**4**) *Method names.* While trace structure, as captured by the attributes above, contributes to inference, method (and class) names also contain useful information about likely mappings, and

we leverage this attribute as well. We use Levenshtein edit distance to compute the similarity of method names. Using this attribute, for instance, we can lend credence to the belief that `setColor()` maps to `parseColor()`.

Our approach combines these attributes to output likely mappings, each associated with a probability. We can rank the results using the corresponding probabilities, and use thresholds to limit the number of results that are output.

STEP 4: Combining inferences across traces. The inference algorithm of Step 3 works by analyzing a single trace pair. In the final step, we combine inferences across multiple traces. During combination, we weight inferences obtained from the analysis of individual trace pairs using the confidence of each inference. Intuitively, more data we have about an inferred mapping from a trace pair, the stronger our confidence in that inference. Thus, our confidence in mapping `A()`→`a()` obtained from a trace pair where these calls occur several times is stronger than the same mapping obtained from a trace pair where these calls occur infrequently. We use this heuristic to combine inferences across trace pairs.

2.3 Framework to Represent and Infer Mappings

We now describe the framework used to represent and infer likely mappings between APIs. We restrict ourselves to identifying likely mappings between methods of the source and target APIs. We do not currently consider the problem of determining mappings between arguments to these methods; that is a related problem that can be addressed using parameter recommendation systems (*e.g.,* [93]). Let $\mathcal{S} = \{\texttt{A}, \texttt{B}, \texttt{C}, \ldots\}$ and $\mathcal{T} = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots\}$ denote the sets of methods in the source and target APIs, respectively. Our goal is to determine which methods in $\mathcal{T}$ implement the same functionality on the target platform as each method in $\mathcal{S}$ on the source platform.

To denote mappings, our framework considers a set of Boolean variables $\mathsf{X}_{\texttt{t}}^{\texttt{s}}$, where $\texttt{s} \in \mathcal{S}$ and $\texttt{t} \in \mathcal{T}$. The Boolean variable $\mathsf{X}_{\texttt{a}}^{\texttt{A}}$ is set to TRUE if the method call `A()` maps to the method call `a()`, and FALSE otherwise.

This framework extends naturally to the case where a method (or sequence of methods) in $\mathcal{S}$ can be implemented with a method sequence in $\mathcal{T}$. For example, suppose that the method `A()` is implemented using the sequence `a();b()` in the target. We can denote this by assigning TRUE

to the Boolean variable X^{A}_{ab}. Likewise, we can also denote cases where the functionality of a sequence of methods `A();B()` in the source platform is implemented using a method `a()` on the target platform by setting the Boolean variable X^{AB}_{a} to TRUE. Although our framework theoretically supports inference of mappings between arbitrary length method sequences (*e.g.,* $X^{ABC\cdots}_{abc\cdots}$ = TRUE), for performance reasons we configured Rosetta to only infer mappings between method sequences of length two.

We approach the problem of inferring API mappings by studying the structure of execution traces of similar applications on the source and target platforms. We use the trace attributes informally discussed in §2.2 to deduce API mappings. Our approach is inherently probabilistic. It cannot conclude whether a call `A()` definitively maps to a call `a()`; rather it determines the likelihood of the mapping. Thus, it infers Pr[X^{A}_{a} = TRUE] for each Boolean variable X^{A}_{a}. As it observes more evidence of the mapping X^{A}_{a} being true, it assigns a higher value to this probability. Therefore, our framework treats each Boolean variable X^{A}_{a} as a random variable, and our probabilistic inference algorithm determines the probability distributions of these random variables.

Each application trace has several method calls, and the inference algorithm must leverage the structure of these traces to determine likely mappings. The inference algorithm draws conclusions not just about *individual* random variables, but also about how they are *related*. For example, consider a trace snippet $\mathsf{Trace}_{\mathcal{S}}$ = (. . .; `A();A();B();B();` . . .) of an application from the source API, and a snippet $\mathsf{Trace}_{\mathcal{T}}$ = (. . .; `a();a();b();b();` . . .) from the corresponding execution of a similar application on the target. Suppose also that these are the only occurrences of `A()`, `B()`, `a()` and `b()` in $\mathsf{Trace}_{\mathcal{S}}$ and $\mathsf{Trace}_{\mathcal{T}}$, respectively, and that these execution traces have approximately the same number of method calls in total. By just observing these snippets, and relying on the frequency of method calls, each of the following cases is a possibility: X^{A}_{a} = TRUE, X^{A}_{b} = TRUE, X^{B}_{a} = TRUE, X^{B}_{b} = TRUE. However, if X^{A}_{b} = TRUE, then because of the relative placement of method calls in these traces (*i.e.,* call context), it is unlikely that X^{B}_{a} = TRUE. Now suppose that by observing more execution traces, we are able to obtain more evidence that X^{A}_{b} = TRUE, then we can leverage the structure of this pair of traces to deduce that X^{B}_{a} is unlikely to be a mapping. Intuitively, the structure of the trace allows us to propagate the belief that if X^{A}_{b} is TRUE, then X^{B}_{a} is FALSE. Our probabilistic inference algorithm uses factor graphs [57, 89] for

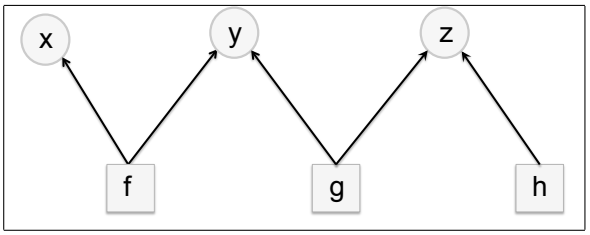

Figure 2.3: Factor graph of $\mathcal{J}$

belief propagation.

Factor Graphs. Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean random variables and $\mathcal{J}(x_1, x_2, \ldots, x_n)$ be a joint probability distribution over these random variables. $\mathcal{J}$ assigns a probability to each assignment of truth values to the random variables in $\mathcal{X}$. Given such a joint probability distribution, it is natural to ask what the values of the *marginal probabilities* $\mathcal{M}_k(x_k)$ of each random variable x_k in $\mathcal{X}$ are. Marginal probabilities are defined as

$$\mathcal{M}_k(x_k) = \sum_{i \neq k, i \in [1..n], x_i \in \{\text{TRUE,FALSE}\}} \mathcal{J}(x_1, x_2, \ldots, x_n)$$

That is, they calculate the probability distribution of x_k alone by summing up $\mathcal{J}(\ldots)$ over all possible assignments to the other random variables. $\mathcal{M}_k(x_k)$ allows us to compute $\Pr[x_k = \text{TRUE}]$. Factor graphs allow efficient computation of marginal probabilities from joint probability distributions when the joint distribution can be expressed as a product of *factors*, *i.e.,* $\mathcal{J}(x_1, x_2, \ldots, x_n) = \prod_i f_i(X_i)$. Each f_i is a factor, and is a function of some subset of variables $X_i \subseteq \mathcal{X}$.

For example, consider a probability distribution $\mathcal{J}(x, y, z)$. Let this distribution depend on three factors $\mathsf{f}(x, y)$, $\mathsf{g}(y, z)$ and $\mathsf{h}(z)$, *i.e.,* $\mathcal{J}(x, y, z) = \mathsf{f}(x, y).\mathsf{g}(y, z).\mathsf{h}(z)$, defined as follows:[1]

$$\mathsf{f}(x, y) = \quad 0.9 \text{ if } x \vee y = \text{FALSE}, 0.1 \text{ otherwise.}$$
$$\mathsf{g}(y, z) = \quad 0.9 \text{ if } y \vee z = \text{TRUE}, 0.1 \text{ otherwise.}$$
$$\mathsf{h}(z) = \quad 0.9 \text{ if } z = \text{TRUE}, 0.1 \text{ otherwise.}$$

A factor graph denotes such joint probabilities pictorially as a bipartite graph with two kinds of nodes, *function* nodes and *variable* nodes. Each function node corresponds to a factor, while

[1] Because $\mathcal{J}$ is a *probability* distribution, the product $\mathcal{J}(x, y, z)$ is in fact $Z.f(x, y).g(y, z).h(z)$, where Z is a normalization constant introduced to ensure that the probabilities sum up to 1. In the rest of this paper, the normalization constant is implied, and will not be shown explicitly.

each variable node corresponds to a random variable. A function node has outgoing edges to each of the variable nodes over which it operates. Figure 2.3 depicts the factor graph of $\mathcal{J}$. The AI community has devised efficient solvers [57, 89] that operate over such graphical models to determine marginal probabilities of individual random variables. We do not discuss the details of these solvers, since we just use them in a black box fashion.

Factor graphs cleanly represent how the random variables are related to each other, and can influence the overall probability distribution. One of the key characteristics of factor graphs, which led us to use them in our work, is that they were designed for belief propagation, *i.e.,* in transmitting beliefs about the probability distribution of one random variable to determine the distribution of another.

To illustrate this, consider the probability distribution $\mathcal{J}(x, y, z)$, discussed above. $\mathcal{J}$ can be interpreted as denoting the probabilities of the outcomes of an underlying Boolean formula for various assignments to x, y, and z. Under this interpretation, we could say that the Boolean formula evaluates to TRUE for those assignments to x, y and z for which the value of $\mathcal{J}(x, y, z)$ is above a certain threshold, *e.g.,* if the threshold is 0.6, and $\mathcal{J}(\text{TRUE}, \text{TRUE}, \text{FALSE})$ is 0.7, we say that the formula is TRUE under this assignment. Now suppose that y is likely to be FALSE, *i.e.,* $\Pr[y = \text{FALSE}]$ is above a threshold. We are asked to find under what conditions on x and z the Boolean formula still evaluates to TRUE, *i.e.,* $\mathcal{J}(x, y, z)$ is above the threshold. From the definitions of the factors f, g, and h, we know that $\mathcal{J}$ obtains values that are likely to exceed the threshold if $x \vee y = \text{FALSE}$, $y \vee z = \text{TRUE}$ and z is TRUE. Given that y is likely to be false FALSE, these factors lead us to deduce that x is also likely to be FALSE (giving $\mathsf{f}(x, y)$ a high value) and z is likely to be TRUE (giving $\mathsf{g}(y, z)$ and $h(z)$ high values), thereby pushing the value of $\mathcal{J}$ above the threshold. Intuitively, the factor graph allows us to propagate the belief about the value of y into beliefs about the value of x and z. §2.4 shows how we cast the problem of inferring likely API mappings using factor graphs, thereby allowing us to transmit beliefs about one mapping into beliefs about others.

2.4 Design and Implementation of Rosetta

We now describe in detail the Rosetta prototype, which currently infers mappings between the Java ME and Android graphics APIs. Specifically, we focus on the machinery that enables Steps 2-4 discussed in §2.2.

2.4.1 Infrastructure for Trace Collection

To record execution traces, we instrumented Java ME programs via bytecode rewriting. We used the ASM toolkit [34] to insert logging functionality that records the name of each method call (and class name), prior to invocation. During runtime, this results in a trace of all methods invoked. We then filter out just those methods that derive from the class `javax/microedition/lcdui` — the Java ME graphics API. In all, this API has 281 distinct methods. Rosetta infers mappings for those methods that appear in application traces.

We did not employ bytecode rewriting for Android applications because of the lack of publicly-available tools to rewrite Android's `dex` bytecode. Instead, we leveraged the Dalvik virtual machine (v2.1r1) to record the names of all methods invoked by an application. We record all method and class names, and then filter methods in the following classes (prefix: `android/`) `graphics`, `text`, `view`, `widget`, `content/DialogInterface`, `app/Dialog`, `app/AlertDialog`, `app/ActionBar`. This API has 3,837 distinct methods. The difference in the sizes of these APIs illustrates in part the difficulties that a programmer manually porting an application would face.

With this infrastructure, a Rosetta user can collect traces for a pair of similar applications on both platforms. As discussed in §2.2, the user must exercise similar functionality in both applications, thereby collecting a pair of traces that record this functionality. This process can be repeated multiple times for the same application, exercising different functionalities, thereby resulting in a database of trace pairs. Although it is hard to provide concrete guidelines to "exercise similar functionality," we found that it was relatively easy to do so for gaming applications. Given similar games, they will likely have the same logic and similar GUIs on both applications. We simply performed the same moves on games in both platforms, avoiding situations that involve randomness where possible (*e.g.,* choosing the two-user mode to avoid the

computer picking moves at random). However, randomness is not always avoidable, *e.g.,* some games only support user versus computer modes, and this randomness may manifest in the corresponding portions of the traces as well. Despite this, Rosetta infers high-quality mappings because its inference algorithm prioritizes method mappings that persist both across the entirety of each trace pair, and across multiple trace pairs.

2.4.2 Trace Analysis and Inference

In this step, Rosetta analyzes each trace pair collected in the previous step and draws inferences about likely API mappings implied by that trace pair. Recall from §2.3 that we use a Boolean random variable $\mathsf{X}^{\mathtt{A}}_{\mathtt{a}}$ to denote a mapping between `A()` and `a()` (likewise $\mathsf{X}^{\mathtt{A}}_{\mathtt{ab}}$ *etc.*, for method sequences). In this step, Rosetta uses factor graphs to compute $\Pr[\mathsf{X}^{M}_{m} = \textsc{true}]$, for each such random variable, where M and m denote individual methods or method sequences from the source and target APIs, respectively. The value of this probability determines the likelihood that the corresponding mapping holds.

The intuition behind Rosetta's use of factor graphs is as follows. The set of random variables X^{M}_{m} implicitly defines a joint probability distribution $\mathcal{J}$ over these random variables: $\mathcal{J}(\mathsf{X}^{\mathtt{A}}_{\mathtt{a}}, \mathsf{X}^{\mathtt{A}}_{\mathtt{b}}, \ldots, \mathsf{X}^{\mathtt{B}}_{\mathtt{a}}, \mathsf{X}^{\mathtt{B}}_{\mathtt{b}}, \ldots \mathsf{X}^{\mathtt{A}}_{\mathtt{ab}}, \mathsf{X}^{\mathtt{B}}_{\mathtt{ab}}, \ldots)$. As in §2.3, we can assign $\mathcal{J}$ a Boolean interpretation. That is, we treat $\mathcal{J}$ as a probability distribution that estimates the likelihood of an underlying Boolean formula being TRUE, under various truth assignments to the random variables $\mathsf{X}^{\mathtt{A}}_{\mathtt{a}}$, $\mathsf{X}^{\mathtt{A}}_{\mathtt{b}}$, *etc*. From this joint distribution, our goal is to find the probability distributions of the individual random variables. Under this Boolean interpretation, if $\Pr[\mathsf{X}^{\mathtt{A}}_{\mathtt{a}} = \textsc{true}]$ acquires a high value, it means that the Boolean formula underlying $\mathcal{J}$ is likely to be TRUE if $\mathsf{X}^{\mathtt{A}}_{\mathtt{a}}$ is TRUE, thereby leading us to conclude that `A()` is likely to map to `a()`. Likewise, if $\Pr[\mathsf{X}^{\mathtt{A}}_{\mathtt{a}} = \textsc{true}]$ acquires a low value, `A()` is unlikely to map to `a()`.

The main challenge in directly realizing this intuition within an inference tool is that the Boolean formula underlying $\mathcal{J}$ is unknown (for if it *were* known, then any satisfying assignment to it would directly yield an API mapping!). As a result, the joint probability distribution $\mathcal{J}$ cannot be explicitly computed. However, the attributes described in §2.2 determine the conditions that are likely to influence $\mathcal{J}$. Thus, we formalize each of these attributes as factors, and estimate the joint probability distribution as the product of these factors. Of course,

Input : A trace pair ($\mathsf{Trace}_{\mathcal{S}}$, $\mathsf{Trace}_{\mathcal{T}}$).
Output : $\Pr[\mathsf{X}_m^M = \text{TRUE}]$ for each X_m^M, where M and m are methods and sequences in $\mathsf{Trace}_{\mathcal{S}}$, $\mathsf{Trace}_{\mathcal{T}}$, respectively.
$\mathsf{MethSeq}_{\mathcal{S}}$ = set of methods and method sequences that appear in $\mathsf{Trace}_{\mathcal{S}}$ (sequences up to length 2 in our prototype)
$\mathsf{MethSeq}_{\mathcal{T}}$ = set of methods and method sequences that appear in $\mathsf{Trace}_{\mathcal{T}}$
foreach ($M \in \mathsf{MethSeq}_{\mathcal{S}}$ and m in $\mathsf{MethSeq}_{\mathcal{T}}$) **do**
  $\mathsf{f}_{\mathsf{freq}}(\mathsf{X}_m^M) = \mathsf{simCount}(M, m, \mathsf{Trace}_{\mathcal{S}}, \mathsf{Trace}_{\mathcal{T}})$
  $\mathsf{f}_{\mathsf{pos}}(\mathsf{X}_m^M) = \mathsf{simPos}(M, m, \mathsf{Trace}_{\mathcal{S}}, \mathsf{Trace}_{\mathcal{T}})$
  $\mathsf{f}_{\mathsf{name}}(\mathsf{X}_m^M) = \mathsf{simName}(M, m)$
foreach ($M, N \in \mathsf{MethSeq}_{\mathcal{S}}$ and m, n in $\mathsf{MethSeq}_{\mathcal{T}}$) **do**
  $\mathsf{f}_{\mathsf{ctxt}}(\mathsf{X}_m^M, \mathsf{X}_n^N) = \mathsf{simCtxt}(M, N, m, n, \mathsf{Trace}_{\mathcal{S}}, \mathsf{Trace}_{\mathcal{T}})$
  $\mathsf{f}_{\mathsf{ctxt}}(\mathsf{X}_n^M, \mathsf{X}_m^N) = \mathsf{simCtxt}(M, N, n, m, \mathsf{Trace}_{\mathcal{S}}, \mathsf{Trace}_{\mathcal{T}})$
Let the set $\mathcal{F}$ denote the factors gathered above.
Let $\mathcal{J}(\mathsf{X}_{\mathtt{a}}^{\mathtt{A}}, \mathsf{X}_{\mathtt{b}}^{\mathtt{A}}, \ldots, \mathsf{X}_{\mathtt{a}}^{\mathtt{B}}, \mathsf{X}_{\mathtt{b}}^{\mathtt{B}}, \ldots \mathsf{X}_{\mathtt{ab}}^{\mathtt{A}}, \mathsf{X}_{\mathtt{ab}}^{\mathtt{B}}, \ldots) = \prod_{f \in \mathcal{F}} f$.
Use factor graph-based inference to obtain marginal probabilities for each X_m^M from $\mathcal{J}$.

Algorithm 1: Inferring likely mappings.

these factors are not comprehensive, *i.e.,* there may be other factors that influence the value of $\mathcal{J}$. Rosetta can naturally accommodate any new factors; they are simply treated as additional factors in the product, and passed to the factor graph solver.

Rosetta's trace analysis computes four families of factors, one each for the four attributes. It combines them and uses them for probabilistic inference of likely mappings as shown in Algorithm 1.

(1) Call frequency ($\mathsf{f}_{\mathsf{freq}}$). The intuition underlying this factor is that if M maps to m (where M and m are individual methods or method sequences), then the frequency with which they appear in functionally similar traces must match. Thus, we compute the relative count of M and m as the number of times that they appear, normalized by the corresponding trace length. We then use the ratio of relative counts of M and m to compute $\mathsf{f}_{\mathsf{freq}}$. This is described in the subroutine simCount, shown in Algorithm 2.

(2) Call position ($\mathsf{f}_{\mathsf{pos}}$). We observed in our experiments that certain API methods and sequences appear only at specific positions in the trace. For example, API methods that initialize the screen or game state appear only at the beginning of the trace. To identify such methods and sequences, we use a similarity metric that determines the *relative position* of the appearance of the method call or call sequence in the trace, *i.e.,* its offset from the beginning of the trace, normalized by the trace length. Of course, there may be multiple appearances of the method

simCount(M, m, Trace$_{\mathcal{S}}$, Trace$_{\mathcal{T}}$)
begin
  relCount(M) = $\frac{\#\text{Occurences of } M \text{ in Trace}_{\mathcal{S}}}{\text{Length(Trace}_{\mathcal{S}})}$
  relCount(m) = $\frac{\#\text{Occurences of } m \text{ in Trace}_{\mathcal{T}}}{\text{Length(Trace}_{\mathcal{T}})}$
  Return min[$\frac{\text{relCount}_{(M)}}{\text{relCount}_{(m)}}$, $\frac{\text{relCount}_{(m)}}{\text{relCount}_{(M)}}$]
end
simPos(M, m, Trace$_{\mathcal{S}}$, Trace$_{\mathcal{T}}$)
begin
  relPos(M)[.] = relative positions of M in Trace$_{\mathcal{S}}$
  relPos(m)[.] = relative positions of m in Trace$_{\mathcal{T}}$
  avgRP(M) = average of values in relPos(M)[.]
  avgRP(m) = average of values in relPos(m)[.]
  if (the values in the array relPos(M)[.] are within a threshold (1% of the trace length) of avgRP(M) and likewise for relPos(m)[.] and avgRP(m)) **then** Return min[$\frac{\text{avgRP}_{(M)}}{\text{avgRP}_{(m)}}$, $\frac{\text{avgRP}_{(m)}}{\text{avgRP}_{(M)}}$]
  else Return UNDECIDED (the value of UNDECIDED is 0.5)
end
simName(M, m)
begin
  if (M and m are individual methods) **then** Return LEVENSHTEIN.RATIO(M, m)
  else Return UNDECIDED
end
simCtxt(M, N, m, n, Trace$_{\mathcal{S}}$, Trace$_{\mathcal{T}}$)
begin
  relCount(M, N) = $\frac{\#\text{Occurences of } M \text{ in preceding context of } N \text{ in Trace}_{\mathcal{S}}}{\text{Length(Trace}_{\mathcal{S}})}$
  relCount(m, n) = $\frac{\#\text{Occurences of } m \text{ in preceding context of } n \text{ in Trace}_{\mathcal{T}}}{\text{Length(Trace}_{\mathcal{T}})}$
  relCount(N, M) = $\frac{\#\text{Occurences of } N \text{ in preceding context of } M \text{ in Trace}_{\mathcal{S}}}{\text{Length(Trace}_{\mathcal{S}})}$
  relCount(n, m) = $\frac{\#\text{Occurences of } n \text{ in preceding context of } m \text{ in Trace}_{\mathcal{T}}}{\text{Length(Trace}_{\mathcal{T}})}$
  if ApproxMatch(relCount(M, N), relCount(m, n)) and ApproxMatch(relCount(N, M), relCount(n, m)) **then** Return HIGH (we configued HIGH to be 0.7), **else** Return (1 - HIGH).
end

Algorithm 2: Subroutines invoked by Algorithm 1.

call or sequence in the trace, so we average their relative positions.

In this factor, we restrict ourselves only to calls and sequences that are localized in a certain portion of the trace, *i.e.,* if the relative positions are not within a threshold (1% of the trace length) of the average, this factor does not contribute positively or negatively to the likelihood of the mapping (UNDECIDED is a probability of 0.5). This is described in the subroutine simPos in Algorithm 2.

(3) Method names (f_{name}). We use the names of methods in the source and target APIs to determine likely mappings. Unlike the other factors, which are determined by trace structure (*i.e.,* program behavior), this is factor relies on a syntactic feature. The simName subroutine in Algorithm 2 uses a ratio based upon the Levenshtein edit distance, computed using a standard Python library [75]. This ratio ranges from 1 for identical strings, to 0 for strings that do not have a common substring. simName only returns a valid ratio for individual methods; for sequences, it returns UNDECIDED.

(4) Call context (f_{ctxt}). We define the context of a method call `A()` in a trace as the set of method calls that appear in the vicinity of `A()`. Likewise, the context of a sequence `A();B()` is the set of method calls that appear in the vicinity of this sequence (if it exists in the trace). Considering context allows us to propagate beliefs about likely mappings. Recall the example presented in §2.3, where considering the frequency of the method calls `A()`, `B()`, `a()`, and `b()` alone does not allow precise inference of mappings. In that example, the context of the calls allows us to infer that if X_b^A is TRUE, then X_a^B is unlikely to be TRUE. Of the four factors that we consider, f_{ctxt} is the only one that relates pairs of random variables; the others assign probabilities to individual random variables.

We define the context of a method call or sequence M in the trace as the set of method calls and sequences that appear within a fixed distance k of M in the trace; in our prototype, k=4. When computing the context of M, we also consider whether the entities its context precede M or follow M. To compute context as a factor, we use the function simCtxt, which considers all pairs of methods and method sequences (M, N) that appear in the source trace, and all pairs (m, n) that appear in the target trace. We then count the number of times M appears in the preceding context of N in the source trace (*i.e.,* within k=4 calls preceding each occurence of N) and normalize this using the trace length (relCount(M,N)); likewise we compute relCount(N,M),

and the corresponding metrics for the target trace. We then check whether relCount(M,N) "matches" relCount(m,n), and relCount(N,M) "matches" relCount(n,m) We do not require the relative counts to match exactly; rather their difference should be below a certain threshold (10% in our prototype); ApproxMatch encodes this matching function.

If both the counts match, then the factor $\mathsf{f}_{\mathsf{ctxt}}(\mathsf{X}_m^M, \mathsf{X}_n^N)$ positively influences the inference that if X_m^M is TRUE, then X_n^N is also TRUE, and vice-versa. The simCtxt function ensures this by returning a HIGH probability value. Likewise, if the counts do not match, $\mathsf{f}_{\mathsf{ctxt}}(\mathsf{X}_m^M, \mathsf{X}_n^N)$ would indicate that X_m^M and X_n^N are unlikely to be TRUE simultaneously. Note that this does *not* preclude X_m^M or X_n^N from being TRUE individually. Intuitively, the Boolean interpretation of $\mathsf{f}_{\mathsf{ctxt}}(\mathsf{X}_m^M, \mathsf{X}_n^N)$ is $(\mathsf{X}_m^M \wedge \mathsf{X}_n^N)$. In our prototype, we set the value of HIGH as 0.7. We conducted a sensitivity study by varying the value of HIGH between 0.6 and 0.8, and observed that it did not significantly change the set of likely mappings output by Rosetta.

To illustrate the context factor, consider again the example from §2.3. There, simCtxt(A, B, a, b) would be HIGH, while simCtxt(A, B, b, a) would be 1-HIGH (using exact matches for relCount instead of ApproxMatch, to ease illustration). Therefore, we can infer that $\mathsf{X}_{\mathtt{a}}^{\mathtt{A}}$ and $\mathsf{X}_{\mathtt{b}}^{\mathtt{B}}$ could both be TRUE, but that $\mathsf{X}_{\mathtt{b}}^{\mathtt{A}}$ and $\mathsf{X}_{\mathtt{a}}^{\mathtt{B}}$ are unlikely to be TRUE simultaneously.

We implemented Rosetta's trace analysis and factor generation algorithms in about 1300 lines of Python code. We used the implementation of factor graphs in the Bayes Net Toolbox (BNT) [66] for probabilistic inference. Rosetta generates one factor for each Boolean X_m^M for each of the three factor families $\mathsf{f}_{\mathsf{freq}}$, $\mathsf{f}_{\mathsf{pos}}$, $\mathsf{f}_{\mathsf{name}}$. Letting S and T denote the number of unique source and target API calls observed in the trace, there are $O(S^2T^2)$ such Boolean variables (because M and m include individual methods and method sequences of length two). Likewise, Rosetta generates two $\mathsf{f}_{\mathsf{ctxt}}$ factors for each pair (M, N) and (m, n), resulting in a total of $O(S^4T^4)$ factors. We restricted Rosetta to work with method sequences of lengths one and two because of the rapid growth in the number of factors. Future work could consider optimizations to prune the number of factors, thereby allowing inference of mappings for longer length method sequences.

2.4.3 Combining Inferences Across Traces

As discussed so far, we apply probabilistic inference to each trace pair, which results in different values of $\Pr[\mathsf{X}_m^M = \textsc{true}]$ for each Boolean variable X_m^M. In this step, we combine these inferences across the entire database of trace pairs. One way to combine these probabilities is to simply average them. However, if we do so, we ignore the *confidence* that we have in our inferences from each trace pair. Intuitively, the more occurrences we see of a method call or sequence M in a source trace, the more confidence we have in the values of $\Pr[\mathsf{X}_*^M = \textsc{true}]$. Therefore, we compute a weighted average of these probabilities, with the relative count of each source call as the weight.

$$\Pr[\mathsf{X}_m^M = \textsc{true}]\Big|_{combined} = \frac{\sum_{Traces} \mathsf{relCount}(M) \times \Pr[\mathsf{X}_m^M = \textsc{true}]}{\sum_{Traces} \mathsf{relCount}(M)}$$

We chose this approach because of its modularity. As we collect more trace pairs and inferences from them, we can combine them with mappings inferred from other traces in a straightforward way using the weighted average approach. Alternatively, we could have chosen to concatenate individual traces (in the same order for both components of each trace pair) to produce a "super-trace," and perform probabilistic inference over this super-trace. However, if we do so, then we would have to reproduce the super-trace after each new trace pair that is collected, and execute the inference algorithm over the ever-growing super-trace. This approach is neither memory-efficient nor time-efficient. In contrast, our weighted average approach provides more modular support to add inferences from new trace pairs as they become available.

This weighted average is presented to the user as the output from Rosetta. We present the output of Rosetta to the user in terms of the inferred mappings for each source API call. For each source API method or method sequence, we present a list of mappings inferred for it, ranked in decreasing order of the likelihood of the mapping.

2.5 Evaluation

2.5.1 Methodology

To evaluate Rosetta, we collected a set of 21 Java ME applications for which we could find functionally-equivalent counterparts in the Android market. In particular, we chose board

games, for two reasons. First, many popular board games are available for both the Java ME and Android platforms. Checking the functional equivalence of two games is as simple as playing the games and ensuring that the moves of the game are implemented in the same way on both versions. Second, the use of board games also eases the task of collecting trace pairs. Moves in board games are easy to remember and can be repeated on both game versions to produce functionally-equivalent traces on both platforms.

To collect traces, we ran Java ME games using the emulator distributed with the Sun Java Wireless Toolkit, version 2.5.1 [73]. For Android games, we used the emulator that is distributed with the Android 2.1 SDK. Table 2.1 shows the games that we used, the number of trace pairs that we collected for each game, and the sizes of the traces for the Java ME and Android versions.

We ran Rosetta on these traces to obtain a set of likely mappings. This set is presented to the user as a ranked list of mappings inferred for each Java ME method (or method sequence). To evaluate the list associated with each Java ME method, we consulted the documentation of the Java ME and Android graphics APIs to determine which members of the list are valid mappings.

2.5.2 Quality of Inferred Mappings

Table 2.2 presents the results of running Rosetta on the traces that we collected. This figure shows the number of distinct Java ME methods that we observed in the trace, grouped by the parent Java ME class to which they belong. Thus, for example, there were four unique Java ME methods belonging to the `Alert` class in our traces, namely, `Alert.setCommandListener`, `Alert.setString()`, `Alert.setType()`, and `Alert.setTimeout()`. In all, there were 83 unique Java ME methods in our traces.

For each of these 83 Java ME methods, we determined whether the top ten ranked mappings reported for that method contained a method (or method sequence) from the Android API that would implement its functionality. As reported in Table 2.2, we found such valid mappings for 59 of these Java ME methods (71%). Figure 2.4 depicts in more detail the rank distribution of the first valid mapping found for each of these 59 Java ME methods. As this figure shows, the top-ranked mapping for 35 of these methods was a valid one (42% of the observed Java ME

Game (#Traces)	Java ME Traces		Android Traces		Factor graphs	
	AvgLen	MaxLen	AvgLen	MaxLen	MaxNodes	MaxEdges
Backgammon (2)	11,523	33,733	214,311	445,861	15,230	13,435
Blackjack (5)	658	1,181	136	369	11,408	10,256
Bubblebreaker (4)	858	2,033	2,366	7,377	2,844	2,358
Checkers (1)	111,790	111,790	1,014	1,014	4,923	4,191
Chess (5)	52,869	259,491	2,278	8,157	7,562	9,664
Four in a Row (3)	4,828	8,764	12,757	23,450	19,868	16,021
FreeCell (4)	12,926	15,271	407	746	128,128	96,096
Hangman (3)	3,715	4,282	3,477	3,481	10,319	11,263
Mahjongg V1 (5)	35,632	150,206	11,494	22,025	8,891	7,062
Mahjongg V2 (5)	3,652	18,321	16,831	49,887	4,703	4,806
Memory (5)	164,809	481,754	19,569	35,382	15,387	16,399
Minesweeper (5)	1,890	5,396	928	1,939	11,675	10,900
Roulette (5)	5,003	6,232	185	227	26,580	19,935
Rubics Cube (3)	16,521	19,343	131	159	21,840	16,380
Scrabble (4)	13,957	14,358	114	184	105,300	78,975
SimpleDice (5)	654	965	581	593	37,152	27,864
Snake (2)	33,399	63,104	11,681	20,372	1,528	1,356
Soltaire (3)	3,436	12,471	8,146	20,805	21,714	20,228
Sudoku (5)	3,897	9,968	17,347	42,567	16,306	24,317
Tetris (2)	486	916	6,116	11,991	13,105	14,520
TicTacToe (4)	154	475	183	418	3,840	4,690

Table 2.1: Statistics of traces and factor graphs for various Java ME and Android games. The actual runtime traces of the games are filtered to leave only Java ME and Android graphics API calls. We report the average and maximum lengths of these filtered traces. For each game, we also show the size of the largest factor graph across all its traces.

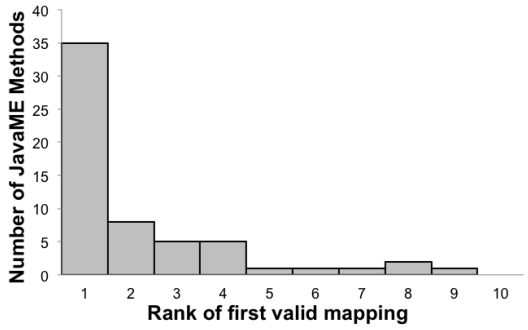

Figure 2.4: This figure shows the rank distribution of the first valid mapping found for each Java ME method. In all, for each of 59 Java ME methods, a valid Android mapping appeared within the top ten results reported for that method. For 35 Java ME methods, the top-ranked mapping was a valid one.

Java ME class	#Methods	#Top-10	(#TotValid)	#Top-1
`Alert`	4	3	(11)	3
`Canvas`	8	5	(18)	4
`ChoiceGroup`	3	0	(0)	0
`Command`	2	2	(5)	0
`Display`	7	5	(13)	3
`Displayable`	6	4	(12)	3
`Font`	5	4	(12)	1
`Form`	4	2	(8)	1
`game.GameCanvas`	5	5	(12)	2
`game.Layer`	3	3	(8)	2
`game.Sprite`	4	3	(13)	2
`Graphics`	21	19	(69)	12
`Image`	4	4	(8)	2
`List`	1	0	(0)	0
`TextField`	6	0	(0)	0
Total	**83**	**59**	**(189)**	**35**

Table 2.2: Results of applying Rosetta to the traces obtained from the games shown in Table 2.1. We have shown the number of unique Java ME methods (categorized by class) for which Rosetta inferred at least one valid mapping in the top ten (#Top-10), and the total number of valid mappings found in the top ten (#TotValid). Also shown is the number of Java ME calls for which Rosetta's top-ranked inference was a valid mapping (#Top-1). See also Figure 2.4 and Figure 2.5 for a more detailed rank distribution of mappings.

methods).

Recall that a Java ME method can possibly be implemented in multiple ways using the Android API. Thus, the ranked list associated with that method could possibly contain multiple valid mappings. Rosetta's output contained a total of 189 valid mappings within the top ten results of the 59 Java ME methods. Figure 2.5 depicts the rank distribution of all the valid mappings found by Rosetta. Below, we illustrate a few examples of mappings inferred by Rosetta:

(**1**) The `Graphics.clipRect()` method in Java ME intersects the current clip with a specified rectangle. Rosetta correctly inferred the Android method `Canvas.clipRect()` as its top-ranked mapping.

(**2**) In Java ME `Graphics.drawChar()` draws a specified character using the current font and color. In Rosetta's output, the sequence `Paint.setColor();Canvas.drawText()`, which first sets the color and then draws text, was the second-ranked mapping for

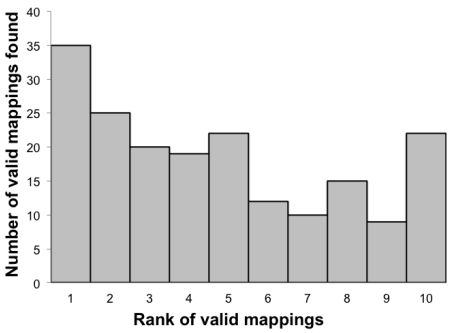

Figure 2.5: This figure shows the rank distribution of all valid mappings found by Rosetta. For each rank, it shows the number of valid mappings that appear at that rank across all Java ME API methods observed in our traces. In all, we found 189 valid mappings ranking in the top ten.

Variant	%Valid
f_{freq}	62.2%
f_{name}	44.0%
f_{pos}	44.0%
$f_{freq} \times f_{ctxt}$	95.6%
$f_{freq} \times f_{name}$	69.8%
$f_{freq} \times f_{pos}$	77.0%

Figure 2.6: Studying the impact of various combinations of factors.

`Graphics.drawChar()`.

(**3**) The `Graphics.drawRect()` Java ME method was mapped to the Android methods `Canvas.drawRect()` (rank 1) and `Canvas.drawLines()` (rank 7), all of which can draw rectangles.

2.5.3 Impact of Individual Factors

Rosetta's output is the result of combining all the four factors discussed in §2.4.2. We also evaluated the extent to which each of these factors contributes to the output. To do so, we constructed factor graphs (using the traces in Table 2.1) individually with f_{freq}, f_{pos}, and f_{name}. Because f_{ctxt} correlates two mappings, we did not consider it in isolation. We also considered a few combinations of factors, to study how the mappings change as factors are added.

We inferred mappings with these factor graphs, and compared the resulting mappings with

those obtained by Rosetta. For each factor graph variant in Figure 2.6, we report the fraction of Rosetta's 189 valid mappings that correspond to a valid mapping inferred by the variant. As we did with Rosetta, we only report valid mappings that rank in the top ten in the output of these factor graph variants. As Figure 2.6 shows, the addition of more factors generally improves the accuracy of the reported mappings. This is because the addition of more factors incorporates more trace features into the probabilistic inference algorithm, thereby removing spurious mappings from the top ten.

2.5.4 Runtime Performance

We ran Rosetta on a PC with an Intel Core2 Duo CPU, running at 2.80Ghz and 4GB memory, running Ubuntu 10.04. For each of the traces reported in Table 2.1, Rosetta took between 2–55 minutes to analyze traces and output mappings. On average, approximately 80% of this time was consumed by the algorithms in §2.4.2 which produce factor graphs, and 20% was consumed by the factor graph solver. The solver consumed 2GB memory for the largest of our factor graphs.

As discussed, Rosetta currently supports inference on method sequences of length up to two. We also configured Rosetta to work with longer method sequences, which would produce larger factor graphs. However, the factor graph solver ran out of memory in these cases.

2.5.5 Experiments with MicroEmulator

There have been some recent efforts [65, 25, 69] to allow the execution of legacy Java ME applications on the Android platform. MicroEmulator [65] is one such open-source tool that works on Java ME applications. It rewrites Java ME API calls in the input application with the corresponding Android API calls. The implementation of MicroEmulator therefore implicitly defines a mapping between the Java ME and Android graphics APIs. However, MicroEmulator does not translate the entire Java ME graphics API, and only allows Java ME applications with rudimentary GUIs to execute on Android devices. Nevertheless, the mappings that it does implement provide a basis to evaluate Rosetta.

We considered a subset of five Java ME games from Table 2.1 that MicroEmulator could

successfully translate (Chess, Minesweeper, Snake, Sudoku, TicTacToe), executed them, and collected the corresponding traces of Java ME API calls. We then repeated the same set of experiments on the MicroEmulator-converted versions of these applications and collected the corresponding traces of Android API calls, and fed these trace pairs to Rosetta. In our evaluation, we determined whether the API mappings inferred by Rosetta contained the mappings implemented in MicroEmulator.

Across the Java ME traces for these five games, we observed 18 distinct Java ME graphics API methods translated by MicroEmulator. In Rosetta's output, we found at least one valid mapping for 17 of these Java ME methods within the top ten ranked results for the corresponding Java ME method. Rosetta only failed to discover an API mapping for the Java ME method `Alert.setString()`. Out of 17 Java ME methods with valid mappings, 8 methods had valid top-ranked mappings to their Android counterparts. MicroEmulator also contains multiple mappings for several of the translated Java ME methods. For the 18 distinct methods in our traces, MicroEmulator contains a total of 26 mappings. Of these, Rosetta's output contained 20 valid mappings within the top ten results for the corresponding Java ME methods.

2.5.6 Threats to Validity

There are a number of threats to the validity of our results, which we discuss now. First, although we attempted to find Java ME and Android games that are functionally equivalent, differences do exist in their implementations. This is because Java ME is an older mobile platform that does not support as rich an API as Android; many Android calls do not even have equivalents in Java ME. Together with randomness that is inherent in certain board games (§2.4.1), this could result in traces which contain Android calls implementing functionality unseen in the source application. They may mislead the attributes used by our inference algorithm (*e.g.,* frequency of calls), leading to both invalid mappings as well as valid mappings being suppressed in the output.

Second, there is no "ground truth" of Java ME to Android mappings available to evaluate Rosetta's output (the mappings in MicroEmulator aside), as a result of which we are unable to report standard metrics, such as precision and recall. We interpreted Rosetta's results by consulting API documentation. Such natural language documentation is inherently ambigious

and prone to misinterpretation. We mitigated this threat by having two authors independently cross-validate the results.

Finally, a threat to external validity, *i.e.,* the extent to which our results can be generalized, comes from the fact that we only inferred mappings using board games. It is unclear how many mappings we would have inferred using other graphical applications (*e.g.,* office applications).

2.5.7 Limitations

In this chapter, we addressed the problem of inferring mappings between API methods of two different mobile platforms using independently developed mobile applications (apps). The intuition behind our approach is that if two apps are implementing the same high-level functionality (*e.g.,* both are TicTacToe games), the developers of the two apps must have used functionally equivalent API methods while implementing the same high-level functionality. If we exercise similar functionality while running the apps, the two apps will take similar execution paths and thus will invoke similar API methods. The traces generated should therfore contain some of the functionally equivalent API methods that map to each other.

While we had some success with this approach with a small set of apps, we realized that it has a number of shortcomings. First, the technique requires independently developed app pairs such that they not only implement the same high-level functionality on two platforms but also should have GUIs with similar visual appearance. During the app search, satisfying the former condition is easy given the huge size of mobile app markets for popular platforms. But satisfying the later condition is time-consuming, since there are quite a large number of different apps, developed by different third-party app developers, all of which have the same high-level functionality but differ in terms of features' richness and GUI. *e.g.,* A simple search for TicTacToe game on Google Play app market and Apple iTunes app store returns over 100 TicTacToe games on each. Picking a pair of TicTacToe games from this dataset of 100x100 games such that the two games have similar GUIs is time consuming. Second, the technique requires the user of Rosetta to manually execute the apps for generating the runtime traces. The state-of-the-art in automatic execution of mobile apps is not advanced enough to drive the execution of apps in controlled fashion. *e.g.,* The best tool available at present to automatically execute Android apps, namely Monkey [23], uses a random sequence of input events rather than

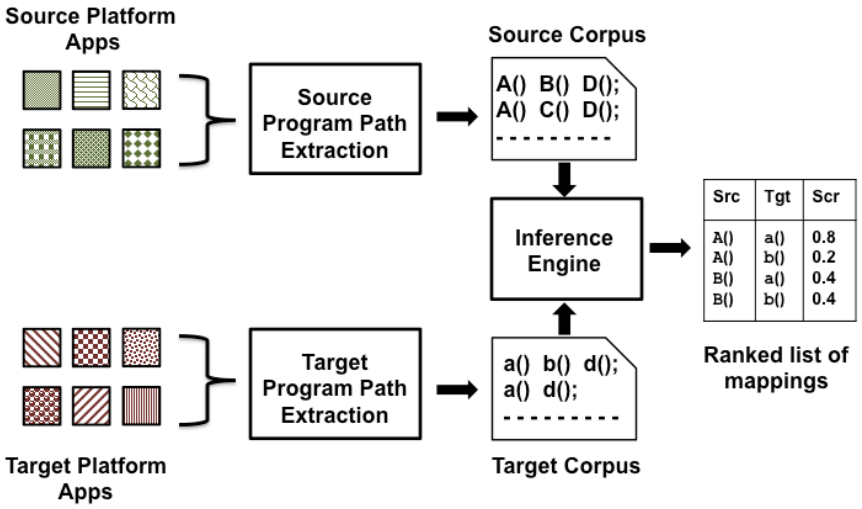

Figure 2.7: Design of the system to identify API mappings using a static approach

allowing the user to feed a specific sequence of input events. Thus, it is practically infeasible to drive the two apps on two different mobile platforms in similar fashion without any user involvement. As a result, this technique has limitations with resepect to the number of apps that can be experimented with. This directly affects the number of mappings that can be inferred, since the inferred mappings are only for those API methods that appear in the traces. The larger the set of apps, more is the number of API methods appearing in the traces and greater is the number of inferred mappings.

In order to overcome these drawbacks, we develop a new technique that relies neither on manual execution of apps nor dynamic analysis to generate the apps' data and does not have the restriction of using functionally equivalent app pairs as the dataset. We describe this technique in the next section.

2.6 Proposing a Static Approach to Infer API Mappings

In this section, we propose a new technique based on static analysis that overcomes the drawbacks of dynamic analysis based technique. We have implemented a prototype tool called Data Driven Rosetta (DDR) [44] that infers mappings between iOS and Android APIs. In theory, the proposed technique has the potential to scale well with the huge number of apps available in the app stores, since it uses static analysis. Our evaluation results are for a small subset of apps and hence not comprehensive. We present the technique as a potential direction to infer

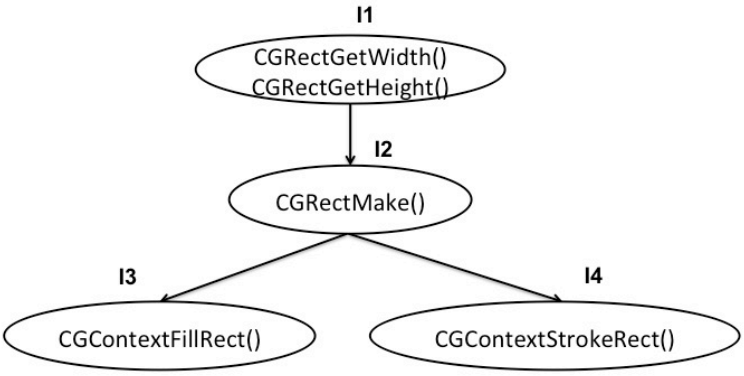

Figure 2.8: Control flow graph of an iPhone app.

API mappings at scale.

2.6.1 Motivation

We propose a technique that is easily scalable to a large number of apps, comparable to the size of the today's app market. Thus emerged our prototype tool, Rosetta (Data-Driven Rosetta), which we propose as a possibly scalable approach to infer API mappings of two mobile platforms. The new approach analyses the apps' data statically using the decompiled binaries of apps. Moreover, it works on as many apps' data as possible, with no particular requirement about the nature of apps (except apps' categories). At a high level, the technique first generates apps' data on both the platforms by analysing the app binaries statically. This data is treated in the same light as the text from two different natural languages. In natural language programming (NLP) literature, there has been a lot of research devoted to deciphering an unknown language or inferring translations between different languages. We use a method employed previously in NLP domain to infer a translation dictionary between non-parallel text corpora of two different natural languages. The output is a set of mappings between API methods of two platforms. We will explain the technique in more detail in section 2.6.2.

2.6.2 Methodology

The overall system design is shown in figure 2.7 tailored to iPhone and Android as the source and target platforms, respectively. We now describe the design in more details.

Our approach takes two sets of apps developed for two different mobile platforms as input.

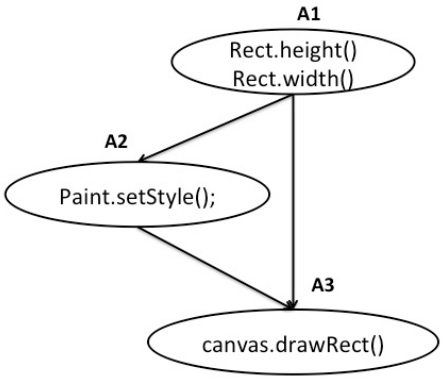

Figure 2.9: Control flow graph of an Android app.

We leverage the notion of category of apps in the mobile app markets to ensure that the fraction of apps belonging to each category is approximately the same across two platforms. This is easy to ensure since app markets of all popular mobile platforms have similar catgories of apps because of expectations from the users to have these apps available across each platform. Using such a dataset of apps helps to ensure that the generated program path data comes from related domains of apps rather than completely unrelated domains.

(1) Extract program paths. Given the apps for two platforms, the next step is to collect the program paths. We first decompile the apps' binaries using the available decompiling tools for the respective platforms to obtain bytecode representation of the apps. We then construct a control flow graph (CFG) from the decompiled bytecode representation using static code analysers for the respective languages. The next step is to traverse the CFG to produce static program paths of apps. For each function in the source code, we first construct its CFG representation that contains only the platform API methods, filtering out everything else. Figure 2.8 shows the generated CFG of an iPhone app. We then convert the CFG into a directed acyclic graph (DAG) by removing back edges. This step ensures that we do not run into infinite number of paths. Here's how the program path generation would work on the CFG given in figure 2.8:

(a) Identify all the entry and exit nodes in the graph. There is a single entry node *I1* and there are two exit nodes, *I3* and *I4*.

(b) For each pair of (*entry*, *exit*) nodes, compute all paths that start at *entry* and end at *exit*. You

***S* (iPhone corpus):**
S1: CGGeometry.CGRectGetWidth() CGGeometry.CGRectGetHeight()
CGGeometry.CGRectMake() CGContext.CGContextFillRect()
S2: CGGeometry.CGRectGetWidth() CGGeometry.CGRectGetHeight()
CGGeometry.CGRectMake() CGContxt.CGContextStrokeRect()

***T* (Android corpus):**
T1: Rect.height() Rect.width() Paint.setStyle() Canvas.drawRect()
T2: Rect.height() Rect.width() Canvas.drawRect()

Note: iPhone and Android classes are prefixed with /System/Library/Frameworks/CoreGraphics.framework/CoreGraphics/ and android/graphics, respectively. For brevity, we only refer to method names and not their classes.

Figure 2.10: iPhone corpus and Android corpus

can use any standar graph algorithm to find all paths between two nodes of a DAG. For the pair (*I1*, *I3*), there is a single path in between them that goes via *I2*. Similarly for the pair (*I1*, *I4*), only one path exists in between them which goes through *I2*.

(c) For each path, print the sequence of API methods inside each of the node, in the same order as the nodes appear in the path. For the given graph, we extracted two node sequences: *I1 I2 I3* and *I1 I2 I4*. These would yield two program paths: *S1* and *S2* as shown in figure 2.10.

Similarly, for Android platform, program paths would be *T1* and *T2* shown in figure 2.10.

(2) Infer mappings. The next step is to feed these program paths composed of sequences of API methods to the inference engine which gives mappings between API methods. We directly rely on the MCCA method [46] as our inference engine which uses a statistical machine learning technique. The MCCA method takes two text corpora, one in source lanaguage and the other in target language as input. The two corpora can be completely unrelated and can be from different domains. At a high level, the method defines a generative model over the observed data which consists of the feature vectors of words in the corpora and the mappings generated so far. The generative model explains the observed data in terms of vectors in a common, hidden space. Their model is based on canonical correlation analysis [26]. If we view the generated source and target API program paths as belonging to two unknown source and target languages, we can feed the program paths to MCCA method to obtain mappings between API methods.

Let *S* and *T* denote the iPhone corpus and Android corpus shown in figure 2.10. We feed

these two corpora to MCCA method. The method outputs a set of mappings between the elements of the corpora which are API methods in our case. In MCCA model, the objective is to maximize the log-likelihood of the observed data (feature vectors and the mappings generated so far). It casts this optimization problem as a maximum weighted bipartite matching problem over a graph consisting of source words and target words. The weight on the edge between a source word and target word denotes confidence in the mapping. The default model outputs at most one mapping for each API method. We modified the default model so as to output top 10 mappings for each source API method corresponding to top 10 weights on edges originating from the given source API method. We consider all non-negative weights on edges, in case the given API method does not have 10 mappings in the final bipartite matching output.

2.6.3 Implementation

We now describe the implementation details of our tool DDR which currently infers mappings between iOS and Android APIs.

iOS program paths generation. For the analysis of iOS apps, we first automatically download iOS apps on a Mac OS device from Apples App Store by using AppleScript [5]. After synchronizing, iOS apps are installed on the iOS device. And then we can get the unencrypted binary code for its analysis using some cracking iOS app tools such as Clutch. In order to disassemble the binary of iOS, we use a popular disassembler called IDA Pro [13]. It supports the Mach-O binary format of iOS executables, but it does not fully provide method names that we need. In Objective-C, method calls are carried out using the dispatch function `objc_msgSend`. The first argument of this function refers to the object name and the second one refers to the method name. However, IDA pro does not resolve the actual targets of method calls to the `objc_msgSend` due to its limitation. Thus, in order to get all the method names of the binary, we use backward slicing [86] which tracks the related instructions backwards. We use IDAPython which is an IDA Pro plugin that allows scripts run in IDA Pro, and also use NetworkX library [17] for graph supports to make CFG. And then we traverse the CFG to make program paths. If necessary, we limit ourselves to generate paths upto a certain length for practical reasons.

Android program paths generation. To analyse Android apps' binaries statically, we need tools capable of processing reverse engineered Dalvik bytecode of apps. It is easier to analyse

Java bytecode though than Dalvik bytecode due to plenty of static anlaysers develoepd for Java. We therefore retarget Android apps' dalvik bytecode to Java bytecode using dare [10] tool. For analysing the resultant Java bytecode, we use a popular static analyser for Java programs called Soot [22]. Soot requires as input either the given program's entry point or the list of classes to be analysed. Because of lack of a single entry point (such as *main()* method in Java programs) in event-driven Android apps, we need to give the list of classes declared in the bytecode as input to Soot. But not all classes in the app's package are of interest to us. An Android app uses a number of external APIs (*e.g.,* various advertisement APIs such as Google Ads, location services API etc.) which are not part of the core platform API and hence can be neglected for analysis purposes. We therefore need to identify the core classes that belong only to the app and not to the external APIs. With the help of android-apktool [1], we first identify the classes belonging only to the app. This list of classes is given as input to Soot. We make use of Soot's API to construct intra-procedural control flow graph (CFG) of functions defined in the app. We then traverse the CFG to produce program paths.

Inference engine. We use MCCA as our inference engine. We adapted MCCA in order to work with the dataset of API methods and modified the source code in many ways. We mention here the major modifications we did: (1) We changed edit-distance function to consider only the method name instead of the entire method signature that forms the *word* in our corpus. (2) We modified the tool so as to output a list of top 10 mappings for each iOS API method. The MCCA tool finds a solution to bipartite graph matching problem. The mappings given as output have a score corresponding to the weight of the edge between the source API method node and target API method node. Instead of emitting a single edge with the highest edge, we find top 10 edges for the same source API method in decreasing order of the edge weights. The mappings associated with such top 10 edges are given as output by our tool DDR. In case the source API method node does not have 10 edges with non-negative weights, we consider all the edges with non-negative edges.

2.7 Summary

In this chapter, we present two approaches to detect similarities between the APIs of a source and target mobile platform. The inferred similarities are in the form of mappings between functionally similar methods or method sequences of the two APIs. We first present a generic approach based on dynamic analysis that uses functionally similar apps on two platforms. Our Rosetta prototype uses this approach to infer likely mappings between the Java ME and Android graphics APIs. As a future direction, we propose another approach based on static analysis to infer mappings between API methods of two mobile platforms. Our prototype tool uses a technique from NLP domain to infer mappings between iOS and Android API methods and can potentially scale to a larger input dataset of apps.

Chapter 3

Detecting Similarities in Mobile Apps

This chapter gives an overview of a technique to detect similarities in mobile apps on a single platform. With the proliferation of mobile apps available on the app markets, plagiarised apps also make their way through these app markets. Plagiarised apps often carry malware with them. They may deprive the developers of original apps from the revenue in the form of in-app purchases or advertisements. Previous approaches proposed by researchers to detect plagiarised apps are based on static analysis. We show that these techniques can be defeated if the process of plagiarising uses strong obfuscation. We present a dynamic analysis based technique that can detect plagiarised apps even in presence of obfuscated code. We show that our approach is effective in detecting plagiarism in mobile apps.

3.1 Introduction

In recent years, smart phone app markets have witnessed explosive growth. Popular app markets, such as those of Apple and Google, now have in excess of a million apps. With such large numbers, and an equally diverse and large developer community, malpractices in app development should come as no surprise. We focus on *app plagiarism*, the practice of using another developer's code, without permission or payment, to build and deploy mobile apps.

Plagiarism has long been a problem in traditional software development. It especially affects developers who wish to protect intellectual property embedded in their software. In response, the community has been actively researching techniques to detect plagiarism ([28], [54], [30], [56], [52], [39], [60]). In the mobile app space, plagiarism is of particular concern. Recent work has shown that plagiarised apps divert advertising revenue as well as the user base from the developers of the original apps [43]. More importantly, most mobile malware are repackaged (*i.e.,* plagiarized) versions of otherwise benign mobile apps [99]. Malicious mobile

apps are often distributed via *third-party app stores*, which lure victims by promising free versions of apps that would otherwise require payment on official app stores. The developers of these mobile apps typically obtain a copy of the original app from the official app store, modify the app with their own malicious functionality, and repackage and distribute it via these third-party app stores. The malicious functionality may include displaying unwanted advertisements to victims, monitoring the activities of victims, or even exfiltrating sensitive data to malicious web sites ([41, 99, 97]).

The rise of mobile malware has also spurred the development of anti-malware tools, a variety of which are available through official app markets. These tools use a number of techniques to detect malware, the most common of which is to use simple signature-based scanning to detect malicious apps. Signatures encode code or data patterns seen in malicious apps but not in benign ones. Anti-malware tools scan a new app upon download, and attempt to match them against their signature database. However, it has long been known in the security community that signature-based malware detection tools can easily be evaded using simple transformations [33]. Such transformations include variable and function renaming and code reordering, which alter the syntactic structure of the code, so that it no longer matches the signatures used by anti-malware tools. Indeed, recent work has shown that most commercially-available tools to detect mobile malware can easily be evaded using simple transformations [85].

In response, there have been a number of efforts to develop techniques that detect malicious mobile apps even in the presence of transformations, focusing primarily on detecting plagiarized mobile apps [98], [97], [35], [47]. To date, these techniques have used static analysis and follow the same basic recipe: Obtain and disassemble a suspect app, and use sophisticated similarily detection algorithms to detect plagiarism. The similarity detection algorithms studied in the literature have primarily been syntactic in nature, ranging from detecting code clones [47], using fuzzy hashing to detect similar code fragments [98], and using machine learning techniques to detect apps that share similar syntactic features [97].

In this paper, we take the position that such static approaches towards detecting plagiarism that rely on syntactic features fundamentally fall short, and can easily be defeated using code obfuscation. We argue that a more robust app plagiarism detection technique is needed, and present such an approach. Our main contributions are two-fold:

(**1**) *Obfuscation defeats syntactic similarity detection.* We show that the use of simple encryption techniques defeats previously-proposed similarity detection tools. These tools rely on syntactic features of the code, and the use of encryption obfuscates these features, making it possible to easily evade detection by these tools. §3.2 presents the results of this study, in which we also study the effectiveness of several off-the-shelf obfuscation tools.

(**2**) *API birthmarks provide robust plagiarism detection.* We develop a robust approach for detecting mobile app similarity. Our approach was inspired by prior work [80] to create unique fingerprints of desktop programs. This approach works by observing the execution of a mobile app—it is therefore dynamic in nature—and recording its interactions with the underlying mobile platform via the API that it exposes. The key idea is that an app can affect the mobile device only by interacting with the platform via its API. Thus, similar apps must interact with the platform in similar ways. We capture "similarity" via the notion of *API birthmarks*, which are subsequences of API calls that are unique to a particular app. In our experiments, we show that the API birthmarks of plagiarized apps substantially resemble those of the original apps. Further, API birthmarks are robust to code obfuscation and encryption because a running app *must* issue API calls to interact with the platform, and these calls themselves cannot be obfuscated. Thus, our approach provides a robust way to detect app plagiarism.

3.2 Obfuscating Mobile Apps

Obfuscating a mobile app is the process of transforming the original source code of the app so that resulting code is hard for humans to interpret or read. App developers obfuscate code for a number of reasons, such as to prevent easy interpretion of the reverse-engineered code, or to protect the code from tampering by others. However, obfuscation can also be used to disguise plagiarism, e.g., when a copyrighted app's source code is being reused without obtaining the appropriate permissions. By obfuscating the plagiarized app, the attacker reduces the probability of the new app being identified as similar to the original app. As we will demonstrate, obfuscated apps can easily evade a number of off-the-shelf code similarity detection tools.

Transformations → Obfuscators ↓	**Renaming**	**Dead code removal**	**Control flow obfuscation**	**String encryption**	**Code encryption**
ProGuard	✓	✓	×	×	×
Allatori	✓	✓	✓	✓	×
DashO	✓	✓	✓	✓	×
Androcrypt	×	×	×	×	✓

Table 3.1: Comparison of different obfuscators in terms of their transformation capabilities. The last row lists the transformations employed by Androcrypt, the obfuscator that we have built.

3.2.1 Comparison of Obfuscators

There are a number of commercial or free off-the-shelf code obfuscators. Some of these obfuscators target only the traditional desktop software, e.g., DashO [11] and Allatori [2], while obfuscators such as ProGuard [20] have been ported to work for Android apps as well. These obfuscators use a number of algorithms to transform the original code:

- **Name obfuscation:** This technique subtitutes randomly-chosen character sequences in place of the original, human-readable names for a number of code artifacts. For example, the obfuscation may transform source code file names, line numbers, field names, method names, argument names and variable names.

- **Control flow obfuscation:** This technique changes the code of an app (either source code or bytecode) to obscure the control-flow structure of the original app. For example, the transformation may target selection and looping constructs and replace them with *goto* statements. When such transformed code is analyzed using a decompiler, it produces code that is difficult to read and understand. Direct jumps may be replaced with indirect ones, which further complicates even basic decompilation tasks, such as constructing a control-flow graph of the program.

- **String encryption:** In this technique, string literals in the code are encrypted and code to decrypt these strings is added to the source code. Examples of such string literals include the text of error and exception messages, and the text of the labels or other GUI components such as dialog boxes, buttons, and drop-down menus.

Table 3.1 shows the transformation techniques used by three off-the-shelf obfuscation tools that can be applied to Android apps. Given the capabilities of these obfuscation tools, we

wanted to evaluate their effectiveness. One way to evaluate obfuscators is to feed the original code and obfuscated code to a code similarity detection tool. Code similarity detection tool would report low similarity for strongly obfuscated code, whereas similarity would be high for weakly obfuscated code. Below we describe our experiment.

We randomly chose ten Android apps from a repository of open-source Android apps [12]. The randomly selected apps and the sizes of the corresponding *.apk* files are shown in the table 3.2. We then applied ProGuard, Allatori and DashO to each of these apps so as to obtain their obfuscated versions. To compare the original app with its obfuscated counterpart, we used two different code similarity detectors: a popular off-the-shelf tool called Androguard [3] and a state-of-the-art code similarity detector, Juxtapp [47], that was developed in an academic setting.

App's Name	**Size of .apk file**
Zxing	720 KB
Connectbot	858 KB
Stardroid	2,188 KB
OpenSudoku	211 KB
Pedometer	46 KB
Reddit	759 KB
Amazed	15 KB
Wikinotes	119 KB
Photostream	134 KB
Mileage	366 KB

Table 3.2: Size statistics of apps chosen for the experiment to compute similarity measures shown in 3.3. The first column shows name of the app and second column gives size of the .apk file.

We observed that the similarity scores reported by both the tools were almost the same. For brevity, Table 3.3 presents the results of this experiment for Androguard (see also 3.6 for experiments with more apps). Androguard reports a number between 0 and 100 to report similarity: a pair of similar apps will receive a score of 100, and a pair of apps with no similarity will get a score of 0. We converted these numbers suitably to a range between 0 and 1. As Table 3.3 shows, off-the-shelf obfuscators are somewhat effective at defeating Androguard's similarity detection algorithm. Among the three obfuscators and ten apps that we tested, we found that Allatori was the least effective at obfuscating apps. We repeated the same experiment on a larger dataset of 50 apps downloaded randomly from the same repository [12] and found

similar results, as shown in the 3.6 .

App Name	ProGuard	Allatori	DashO	Androcrypt
Zxing	0.61	0.93	0.56	0
Connectbot	0.60	0.76	0.57	0
Stardroid	0.53	0.82	0.54	0.51
OpenSudoku	0.83	0.82	0.59	0.01
Pedometer	0.84	0.67	0.52	0.03
Reddit	0.47	0.94	0.37	0
Amazed	0.44	0.89	0.75	0.1
Wikinotes	0.92	0.86	0.67	0.28
Photostream	0.77	0.92	0.64	0.03
Mileage	0.72	0.85	0.56	0.56

Table 3.3: Results of similarity measure (between 0 to 1) reported by a popular similarity detection tool, Androguard. the Android app's binary executable file. Each column corresponds to app pairs in which obfuscated apps have been obtained by running the corresponding obfuscator.

3.2.2 Androcrypt: An Encrypting Obfuscator for Android Apps

Given the results with the three off-the-shelf obfuscation tools, we asked whether it was possible to build an obfuscator that would be even more effective at transforming an app, so that tools such as Androguard would be rendered ineffective. Drawing on the ideas used to create polymorphic and metamorphic malware [92], we built *Androcrypt*, an encrypting obfuscator for Android apps. Androcrypt takes a *.apk* file corresponding to an Android app as input, encrypts the app and packages it as the payload for a new, obfuscated app.

In more detail, Androcrypt operates as follows. Every Android project consists of a collection of files and directories, in which source code files, binary files and resource files are organized into directories such as *src*, *bin*, *res*, *assets* etc. Typically, raw data files are stored in the *assets* directory which then can be read as a byte-stream using the *android.content.res.AssetManager* class in Android. *AssetManager* class provides lower-level API to open and read the raw files. When supplied an input *.apk* file as input, Androcrypt first creates an empty Android project with all the relevant directories. It then uses the Java cryptography library (we used the *AES/CBC/PKCS5Padding* mode) to encrypt the input *.apk* file, and places it in the *assests* directory. Androcrypt incorporates a new *Activity* class in the new Android project that first reads the encrypted app stored in *assets* directory using *AssetManager*

API and decrypts it using the Java cryptography library. The *Activity* then dynamically loads the classes from the DEX bytecode of the decrpyted app using the *dalvik.system.DexClassLoader* class. Androcrypt replaces all the original source code files of the application that reside in *src* directory with the single source file of the above *Activity* class. This Android app is then packaged as an *.apk* file and distributed as the obfuscated app. When the app is started, the first component that executes is the new *Activity* class, which decrypts the original app and loads its classes.

While the steps described above suffice to obfuscate a majority of Android apps, there are a few categories of apps that fail to start up when obfuscated this way. Such apps contain classes inherited from either one of two Android classes: *android.content.ContentProvider* or *android.app.Application* In Android, the *ContentProvider* class manages the sharing of data between multiple apps, and is used by apps that share data with other apps. For example, a texting app might use this class if it shares data with the contacts list on the phone. Likewise, the *Application* class is used by Android apps to store any global application state. An Android application using any one of these two classes declares its use in its manifest file. When the Android runtime system executes an Android app, it scans the manifest to determine whether these classes are being used by the app, and first loads these classes before launching the main *Activity* class.

Androcrypt packages the original app and includes the decryption functionality in the main *Activity* class of the repackaged app, which must start first. As a result, the obfuscator cannot extract the individual *ContentProvider* or *Application* classes, thereby breaking the app's functionality. For such apps, Androcrypt uses a hybrid strategy: it obfuscates the *ContentProvider* and *Application* classes using ProGuard, while using encryption on the rest of the *.apk* file.

	ProGuard	**Androcrypt**
Mean	0.68	0.07
Median	0.72	0.02

Table 3.4: Mean and median of the similarity measures (between 0 to 1) reported by Androguard for the dataset of 53 apps. Each column corresponds to app pairs in which obfuscated apps have been obtained by running the corresponding obfuscator.

The last column of Table 3.3 shows the results of obfuscating apps using Androcrypt. We

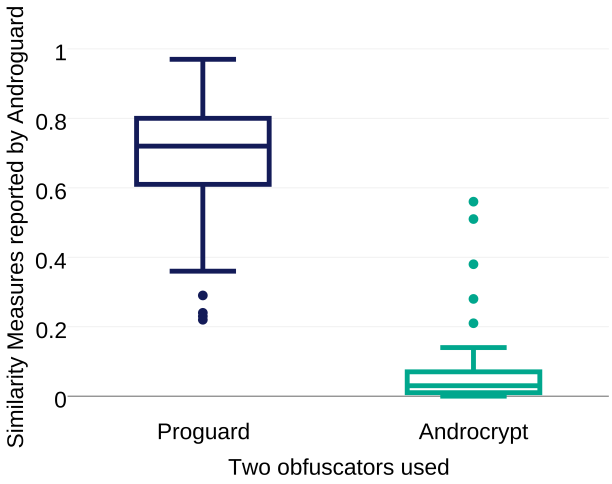

Figure 3.1: Distribution of similarity measures reported by Androguard when two different obfuscators were used. Box on left corresponds to using ProGuard as the obfuscator whereas box on right corresponds to use of Androcrypt as obfuscator. The values used to draw the plots are taken from the columns labeled as (PG, AG) and (AC, AG) in 3.6.

observe that majority of the apps had low similarity scores when Androcrypt was used as compared to any of the three obfuscation tools. Two exceptions were *Stardroid* and *Mileage* apps. These two apps used classes inherited from the *ContentProvider* and *Application* classes; these classes were not encrypted, which explains why these apps had a higher Androguard-similarity score than the other apps. Thus, Androcrypt's encryption-based approach to obfuscating Android apps is more effective than the techniques used by ProGuard, Allatori and DashO.

We repeated the same experiment on a larger dataset of 53 apps. The complete results are given in 3.6. For brevity, we report the mean and the median of the scores in Table 3.4, and the box plots of the scores is shown in Figure 3.1. As is evident from the plot, the similarity measures reported by Androguard on app pairs in which Androcrypt was used as the obfuscator dropped significantly as compared to the values reported on the same app pairs when off-the-shelf obfuscator, ProGuard, was used. The results demonstrate that simple encryption as used in Androcrypt can easily defeat existing similarity detectors.

3.3 Methodology

3.3.1 API Birthmark

There are many off-the-shelf program transformation tools available that can modify the source code of the program without affecting the program's functionality. As discussed in section §3.2, obfuscators are capable of transforming the program so as to evade a popular similarity detection tool for Android.

More generally, a large majority of existing similarity detection tools that have been proposed in the research literature (for Android) rely on simple static properties of the program. Such static properties can be, for example, a unique hash of the executable binary code ([98]), a feature vector comprising the set of permissions requested by the application and the set of API methods present in the source code of the application ([97]). An encrypting obfuscator such as Androcrypt completely transforms the syntatic structure of the application, so that any attempt to recover such static properties can be defeated by suitably encrypting the files of the application. Such transformed programs can easily be passed off as the originals, thereby allowing app plagiarism.

In this paper, we develop a robust approach that detects plagiarised mobile apps. Our approach was inspired by an effective technique to uniquely identify desktop programs by creating their *software birthmarks* [83] dynamically. Such a birthmark represents a unique fingerprint of the program that characterizes its runtime behavior. Two programs that have the same birthmark are likely to implement similar functionality, and are likely to have originated from the same source code.

In this approach, birthmarks are *dynamic in nature* and are computed by observing the runtime behavior of the program. As long as there are no significant high-level changes in the behavior of the original and the obfuscated program, the dynamic birthmark of both versions will be similar. Thus, dynamic birthmarks are more robust to program transformation attacks and are more likely to be preserved during obfuscating transformations.

Since a dynamic birthmark is determined by the runtime behavior of the program, it is important that the birthmark captures program properties that constitute the core functionality of the program. If the program's functionality changes, the birthmark of the resulting program

must be distinct from that of the original version. However, minor changes in the functionality of the program, as would be expected when an attacker adds new functionality to a plagiarized program, must not cause large deviations in the value of the birthmark.

A desktop program interacts with the environment in which it is run (*i.e.,* operating system) to meet the desired functionality. Similarly, an Android app must interact with the Android and Java runtimes in order to achieve functional goals. These interactions are in the form of the Android and Java API methods invoked by the app. We log these method invocations and collect them in a trace when the app is executed. We leverage prior work (Schuler *et al.* [80]) to define an API birthmark of the app over the sequence of method invocations by the app. Any new functionality introduced into the app (or old functionality deleted from the original) will likely introduce changes into the sequences of API methods invoked by the app and hence will result in a different birthmark.

3.3.2 API Birthmark Algorithm

We now describe in more detail the birthmark-based approach for plagiarism detection. Our approach works on a pair of Android apps, and uses birthmarks to compute a similarity coefficient between 0 and 1 (as was the case with Androguard). As explained in §4, we use Jaccard index to calculate the similarity coefficient. A value closer to 1 indicates high similarity in the observed behavior of apps where as a value closer to 0 signals low similarity in the observed behavior. Although we describe the approach for Android apps, we hypothesize that it will be applicable to other mobile platforms as well. In the description below, A is an unmodified Android app, and A_{obfs} is an app that we suspect is an obfuscated variant of A.

3.3.2.1 Trace Collection

The first step is to run both the apps A and A_{obfs} independently, exercising as much functionality as possible, and collect the execution traces. We then filter the trace so as to retain only those method calls that are invoked on objects whose classes belong to the Android API. We do this filtering because we are interested in observing only the interaction between the app and the underlying Android API. Figure 2.2 shows snippets Trace_A and $\mathsf{Trace}_{A_{obfs}}$ of two traces obtained by running A and A_{obfs} respectively, by using the Monkey tool [23] as described in

3.5.10 and filtering the corresponding traces. We will use these snippets in our explanation below, although our approach works on the entire trace.

3.3.2.2 Computing the Similarity Coefficient

In this step, we compute whether the two apps A and A_{obfs} are similar, using the two traces Trace_A and $\mathsf{Trace}_{A_{obfs}}$. This step can be broken down into four sub-steps.

(**1**) *Collecting object-level API method calls.* In an object-oriented language such as Java, a trace of a program is a global sequence of API method calls invoked on objects in the program. We split this global sequence into different *object-level* sub-sequences. Each one of such sequences contains all the API method calls that were invoked on a single object. Such a division avoids to a certain extent the effect of reordering of method calls between different traces introduced by changes in thread scheduling in multi-threaded programs [80].

Let us now turn to the sample trace snippets Trace_A and $\mathsf{Trace}_{A_{obfs}}$ to compute object level API method calls. In these two snippets, we will assume that there is only one object of each of the class types `Activity` and `View`. In general, there could be multiple objects of the same class type, and the sequence of function calls will be collected for each of them individually, based on the object IDs. Let's use the class name itself as the object ID for further simplicity. We accumulate the sequence of function calls invoked on each of the two objects individually. Thus, we get two method sequences each from each of the two traces Trace_A and $\mathsf{Trace}_{A_{obfs}}$. Each of the method calls is prefixed with the full class name such as `android/app/Activity` and `android/view/View` in our actual calculation. We are omitting the class name prefix here for brevity.

API method calls grouped together by objects in Trace_A

`Activity.onCreate();`
`Activity.setContentView();`
`Activity.findViewById();`

`View.setVisibility();` `View.setOnClickListener()`

API method calls grouped together by objects in $\mathsf{Trace}_{A_{obfs}}$
`Activity.onCreate();`
`Activity.setContentView();`
`Activity.findViewById();`
`Activity.getWindow()`
`View.setVisibility();`
`View.setOnClickListener()`

(**2**) *Generating k-length sequences.* The object-level API method call sequences are long and hence are not easy to compare between different program runs. Schuler *et al.* [80] proposed the idea of chopping up these sequences using a sliding window to generate a set of smaller method sequences, and we use this idea as well. Let us assume that for the sample trace snippets Trace_A and $\mathsf{Trace}_{A_{obfs}}$, the length of sliding window is 2. So we can break up the object-level sequences obtained in the above step into a set of sub-sequences, each of length 2 as shown in the table below. From here on, we are omitting the class names which we use in our actual computation.

2-length sequences from Trace_A
`onCreate(); setContentView()`
`setContentView(); findViewById()`
`setVisibility(); setOnClickListener()`

2-length sequences from $\mathsf{Trace}_{A_{obfs}}$
`onCreate(); setContentView()`
`setContentView(); findViewById()`
`findViewById(); getWindow()`
`setVisibility(); setOnClickListener()`

(**3**) *Calculating birthmarks.* Finally, we compute birthmark as the union of all 2-length sequences of all objects. The following table shows the birthmarks computed for the two apps A and A_{obfs}. In this case, the union operation merely combines the two sets of sequences without any deletions. But in general, the union will result in deletions of common sequences present among the different sets.

Birthmark B_A for the app A
`onCreate(); setContentView()`
`setContentView(); findViewById()`
`setVisibility(); setOnClickListener()`

Birthmark $B_{A_{obfs}}$ for the app A_{obfs}
`onCreate(); setContentView()`
`setContentView(); findViewById()`
`findViewById(); getWindow()`
`setVisibility(); setOnClickListener()`

(**4**) *Computing similarity coefficient.* We use Jaccard index [15] as a measure of similarity between two apps. Given two birthmarks B_A and $B_{A_{obfs}}$ of two apps A and A_{obfs}, the similarity coefficient between two apps is therefore given by Jaccard index of the two sets B_A and $B_{A_{obfs}}$.
$Sim(A, A_{obfs}) = \frac{|B_A \cap B_{A_{obfs}}|}{|B_A \cup B_{A_{obfs}}|}$
Thus, for our running example of the two apps, the similarity coefficient is equal to:
$Sim(A, A_{obfs}) = \frac{3}{4} = 0.75$. Once we have computed the similarity coefficient between two apps, we use a threshold to decide whether the two given apps should be categorized as plagiarized. Explanation of how a threshold is chosen is given in section 3.5.3.

3.4 Implementation

Our implementation of the birthmark-based similarity detection approach consists of two parts. The first part is the system that profiles applications and collects execution traces and is described below. The second part is the implementation of the birthmark algorithm described in the previous section, and we do not describe it in further detail here.

The Android SDK ships with a default method profiling tool [8], which can be used to collect execution traces of an application running either on an Android device or on the Android emulator. We used this tool to collect run-time traces of apps executing in the Android emulator. We worked with Android SDK 2.3.7 (Gingerbread). In this section, we will describe the changes we made to the Android framework for trace collection and filtering.

- *Using default Android profiler:* Dalvik Debug Monitor Server (DDMS) [8] is a debugging tool provided in Android's SDK that also offers the ability to trace apps. Users can leverage DDMS to trace the execution of a running app as it performs arious activities using a simple user interface. The resulting file is a concatenation of data in binary format and textual information about mappings between the binary identifiers in the data and the corresponding method names [9]. We wrote a C program to parse the trace file

Input : A trace T consisting of a sequence of (*thread-id*, *method-signature*, *object-id*) entries
Output : Birthmark *B* containing a set of sequences of function calls
Assign window-length = 4
L_{obj} = list of sequences, each of length window-length, of function calls invoked on a particular object *obj*
C_{obj} = current sequence, whose length ¡= window-length, of function calls invoked on a particular object obj
$len_{C_{obj}}$ = length of C_{obj}
foreach entry (*thread-id*, *method-signature*, *obj*) in the trace **do**
  Add *obj* to the set of unique object-ids *uniq-objs*
foreach *obj* in *uniq-objs* **do**
  Initialize the list L_{obj} to empty
  Initialize current sequence C_{obj} to empty
  Initialize current sequence's length $len_{C_{obj}}$ to be zero
foreach entry (*thread-id*, *method-signature*, *obj*) in the trace **do**
  Append *method-signature* to C_{obj}
  Increase value of $len_{C_{obj}}$ by 1
  if $len_{C_{obj}}$ == window-length **then**
    Add the sequence C_{obj} to the list L_{obj}
    Decrease value of $len_{C_{obj}}$ by 1
    Delete the first element from C_{obj}
foreach *obj* in *uniq-objs* **do**
  if $len_{C_{obj}} > 0$ **then** Add the sequence L_{obj} to the list L_{obj}
Initialize *birthmark* to empty set {}
foreach *obj* in *uniq-objs* **do**
  birthmark = union of (*birthmark*, L_{obj})

Algorithm 3: Computing Birthmark

and output the sequences of methods invoked in the trace.

- *Modifications to the profiling code:* We modified the source code of the default profiler to make two changes to the trace generation:

  (**1**) *Tracing app-specific method calls*. The default profiler in Android profiles all the method calls in the call hierarchy including those invoked by the application as well as the methods executed by the underlying Android framework and the Dalvik virtual machine. We are interested in logging only the method calls invoked by the application. So we modified the source files of the Android framework, so that the profiler logs a method call, only if the return address of the calling function falls within the memory boundaries at which the application is loaded. This ensures that the method call being logged was

Input : Birthmarks B and B_{obfs} of two apps, A and A_{obfs} respectively
Output : A similarity coefficient between the two apps, A and A_{obfs}
$I = B \cap B_{obfs}$
$U = B \cup B_{obfs}$
Similarity Score = $\frac{|I|}{|U|}$

Algorithm 4: Calculating Similarity

invoked from within the application and does not include method calls invoked outside from the application such as those made by the Android API.

(**2**) *Emitting object ID*: For calculating API birthmarks, we need the ID of the object on which each method call is invoked during the application run, so as to segregate API method calls based on the respective invoked objects. Hence we modified the Android source so as to emit the object ID in each record of the generated trace. We log the following fields in the final trace: *thread ID, method name, method arguments and return type, class name, object ID*.

- *Trace Filtering:*

  There are three types of APIs that an Android application interacts with:

  (**1**) Android framework API: Set of methods provided by the underlying Android framework

  (**2**) Java standard API: Set of methods provided by the standard Java language implementation

  (**3**) APIs that are part of the application package (including various advertisement libraries)

  During birthmark computation, we want to include only those method calls that are indispensable to the Android application. Hence we decided to log the interaction of the application with the underlying Android framework. We decided not to keep track of the interaction of the application with the libraries that come as part of the application package (such as advertisement libraries). Such APIs being internal to the application, are easy to replace with other equivalent APIs. For example, consider a free mobile application that uses multiple advertisement libraries to display advertisements to the users.

An attacker can easily remove one of these APIs and repackage the application, without changing the application's functionality. Also, one can easily change the methods' names declared inside one of these APIs. The discussion about exclusion of Java API methods is given later in Section 3.5.8.

3.5 Evaluation

3.5.1 Goals

To evaluate the API birthmark algorithm, we need to answer the following question: What are the essential characteristics of a birthmark of a mobile app?

A birthmark should be able to detect high similarity between *identically behaving* copies of the same program, even if the source code of the two programs differs significantly (*e.g.,* because of applying various program transformations). As a corollary, if a large portion of the source code of two programs is the same, the birthmark should detect them as copies. Additionally, when given two dissimilar programs as input, it should be able to distinguish between them by giving low value of similarity. We conducted different experiments to evaluate the API birthmark algorithm on these three factors. We first present our evaluation setup which is followed by a description of these experiments.

3.5.2 Evaluation Setup

The very first step in our evaluation was creation of the required dataset of app pairs. To assess the resilience of the birthmark against program transformations, we need a dataset consisting of two types of apps, a corpus of apps that implement a variety of functionalities, and the same apps, obfuscated using semantics-preserving transformations on the original apps. As explained in §3.2, we designed our own obfuscator, named Androcrypt, that would encrypt the app's binary, store the decryption logic as the first statement to be executed in the new app and thus produce an identically-behaving obfuscated app. We collected a corpus of Android apps, and obfuscated them using Androcrypt. We then ran the API birthmark algorithm on execution traces of all possible (A, $A'_{obfuscated}$) pairs of apps from this dataset. Each pair consists of an original app A and an app $A'_{obfuscated}$ obtained by obfuscating an original app A'. Here, two

cases are possible: either $A^{'} = A$ *i.e.,* $A^{'}$ is the same app as A or $A^{'} \neq A$ *i.e.,* $A^{'}$ is a different app than A. In the first case when we run the algorithm on traces of (A, $A_{obfuscated}$) types of pairs, consisting of an original app A and its obfuscated counterpart $A_{obfuscated}$, we are testing the resilience of the API birthmark against obfuscations. In the second case when pairs are of type (A, $A^{'}{}_{obfuscated}$) where $A^{'}{}_{obfuscated}$ is obfuscated counterpart of an app $A^{'}$ which is *different* than A, we are testing the ability of API birthmark to distinguish between distinct apps.

To compute the API birthmark of an original app A or its obfuscated counterpart $A_{obfuscated}$, we need the corresponding runtime trace. Executing each app manually is time consuming and would impede the detection of plagiarized apps in a large collection (e.g., at app market scale), thereby limiting the scalability of the approach. Therefore, we decided to automate the process of Android app execution. For running Android apps without any manual intervention, we used a tool called Monkey [23] that drives the app automatically by generating input events such as clicks and touches, as described in §3.5.10. We then collected the execution trace using DDMS [8] as explained in section 3.4. From our dataset of (A, $A^{'}{}_{obfuscated}$) pairs of original apps and their obfuscated counterparts, the automatic execution succeeded for a total of 350 app pairs (downloaded randomly from Google's official Android app market).

We divided our dataset into two, a smaller set of 50 apps was used as the *training set* and the remaining apps formed an *evaluation set*. The *training set* was used to set the threshold for the similarity measure. Using this threshold, we evaluated the birthmark algorithm on the apps in the *evaluation set*. We also did a number of experiments on this dataset such as verifying the credibility of the API birthmark.

All our experiments were done on Linux machine with Intel i5 quad-core 3.10GHz processor, 8 GB RAM and running Ubuntu 12.04. On an average, it took 0.2 seconds to filter two traces and run API birthmark algorithm on one pair of apps.

3.5.3 Choosing a Threshold Value

Prior to performing the experimental evaluation, we first need to set the threshold for the similarity coefficient that is used to determine whether an app is plagiarized. As described in §3.5.2, we used the smaller training set of 50 apps to set the threshold. The 50 original apps and 50 obfuscated counterparts of these apps formed 50×50 (A, $A_{obfuscated}$) app pairs. On every pair in

this set, we ran the API birthmark algorithm. We experimented with different values of threshold and calculated the number of false positives and false negatives, as reported in 3.5. The total number of wrong classifications is equal to the sum of false positives and false negatives, as shown in the last column of 3.5. One could choose to minimize the number of false positives alone or the number of false negatives alone. We chose to minimize the total number of wrong classifications, and hence decided to set the threshold value to 0.5, which gives the least number of wrong classifications as shown in the last column of 3.5.

Threshold	**False negatives**	**False positives**	**False negatives + false positives**
0.2	2	87	89
0.3	5	46	51
0.4	13	9	22
0.5	16	2	18
0.6	18	2	20
0.7	21	0	21

Table 3.5: Evaluation for different values of threshold

We used this threshold to evaluate the API birthmark algorithm on the apps in the evaluation set.

3.5.4 Setting the Window Size

Another parameter to be set for the API birthmark algorithm is the window size that is nothing but the length of the API method sequences generated from the execution traces during the API birthmark calculation, as explained in section 3.3. We used the training set to experiment with different windows sizes and calculated the wrong classifications done by the API birthmark algorithm. Table 3.6 shows the results.

These values show that choosing a smaller window size increases the number of false positives, which implies that the similarity between different programs increases where as a bigger window size leads to a rise in the number of false negatives, which suggests that the similarity between identical programs decreases. We chose to set the default window size to 3, so that the API birthmark algorithm can distinguish between different programs (less false positives), at the same time not having an unacceptable number of false negatives.

Window length	False negatives	False positives	False negatives + false positives
1	3	28	31
2	6	17	23
3	16	2	18
4	18	2	20

Table 3.6: Evaluation for different window lengths

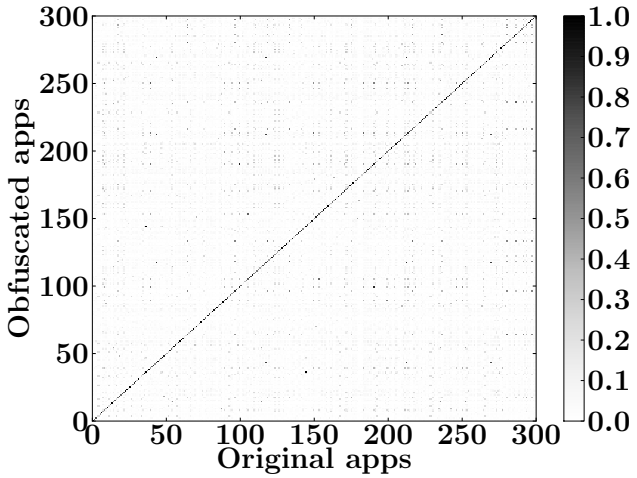

Figure 3.2: API birthmark results on pairwise comparisons for 300 apps (300x300 pairs). Diagonal shows the result of comparing an app and its obfuscated counterpart. Non-diagonal shows comparisons between app and obfuscated counterpart of a different app.

3.5.5 Detecting Obfuscated Apps

We have 300 original apps and 300 obfuscated apps produced by using the encrypting obfuscator described in section 3.2. The total number of possible (A, $A_{obfuscated}$) pairs is therefore 300×300. We run the API birthmark algorithm for every pair, thus producing a matrix of dimensions 300×300. The results are shown in figure 3.2. A point on the diagonal point corresponds to similarity measure between an app and the obfuscated version of the same app, and therefore should have a value closer to 1. A non-diagonal point corresponds to similarity measure between an app and the obfuscated version of a different app, and should therefore have a value closer to 0.

We evaluated the similarity measures produced by the API birthmark by calculating the

Total apps	False negatives	False positives
300	45 (15%)	61 (20.3%)

Table 3.7: Evaluation of API birthmark algorithm for the results in Figure 3.2

number of false positives and false negatives as shown in table 3.7. The number of false negatives is nothing but the number of apps, A, for which the similarity coefficient produced by API birthmark for the pair (A, $A_{obfuscated}$) is less than the threshold and $A_{obfuscated}$ is the obfuscated counterpart of the same app as A (we chose a threshold value of 0.30; we discuss the computation of this threshold in §3.5.3). The number of false positives is the number of apps, A, for which similarity coefficient produced by API birthmark for the pair (A, $A_{obfuscated}$) is greater than the threshold and $A_{obfuscated}$ is the obfuscated counterpart of a different app than A.

There are three cases in which the dynamic API birthmark reports a large similarity coefficient between apps of a certain category in spite of the apps being distinct.

(**1**) **Customized apps:** It is a common practice among app developers to release the same app multiple times, each one built under different package name, such that the core functionality of all packages is the same but the input configuration files are different for each package. An example would be different packages of the app, each one built to display the text in the app in a different language. Such apps have identical source code and differ only in the resource files such as text files or image files. As a result, the execution traces generated during execution of two such apps are nearly identical. API birthmark algorithm therefore detects high similarity between such apps. In our dataset, we found 16 apps in total that fall under this category. An example of one app is given below. This app is a puzzle game, released under three different packages, each one showing different quiz questions, but having identical structure.

App name	Package name
How I Met Your Mother Trivia	com.pbgames.q.himym
Gossip Girl Trivia	com.pbgames.q.gg
Two and a Half Men Trivia	com.pbgames.q.tahm

The classification of packages of two such apps as similar by the API birthmark is indeed truthful. Therefore, we removed such cases from the count of false positives. The table below

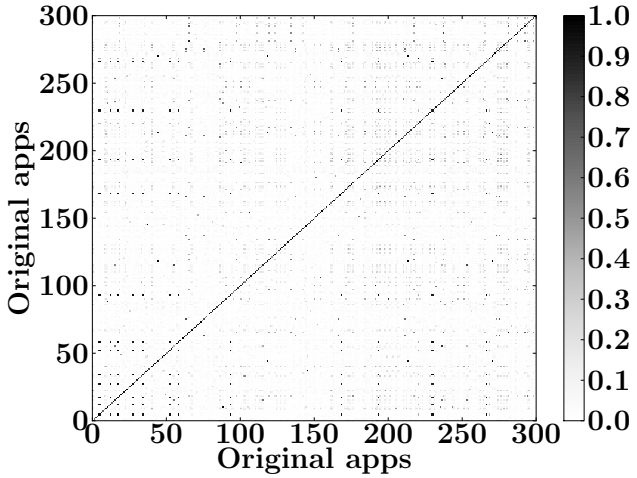

Figure 3.3: API birthmark results on pairwise comparisons for 300 apps (300x300 pairs). Diagonal shows the result of comparing two traces of the same app. Non-diagonal shows comparisons between traces of two different apps.

gives the number of false positives after this filtering.

Total apps	**False negatives**	**False positives**
300	45 (15%)	45 (15%)

(**2**) **Use of programming framework:** There are many programming frameworks such as PhoneGap [19] that let developers write apps using web technologies. These frameworks interact with the underlying mobile operating system such as Android. Hence apps developed using such frameworks exhibit a common set of API method sequences that are part of this interaction. On encountering this common set of method sequences in the traces of such apps, API birthmark algorithm computes high similarity between them. To prevent such apps from being detected as similar, we can collect the set of such commonly found API method sequences and discard them during API birthmark calculation.

(**3**) **Use of common libraries:** Many Android apps use a common set of libraries, such as Google's advertisement library. Because of the interaction of the advertisement library with Android, such apps also display a common API methods during their execution. A similar approach as mentioned above can be used to filter out the common method sequences and thus prevent the classification of the apps as similar.

We believe that the above two filtering techniques would further reduce the number of false positives observed.

3.5.6 Detecting Identical Apps

The goal of this experiment is to test the API birthmark's ability to detect copies of the same app. Since a birthmark of an app is its unique fingerprint, birthmarks of two identical copies of the app (such copies have the same source code too) should be the same. We executed each app twice, treating the resulting two traces as if generated by identical copies of the same app. For each of the two executions of the app, we gave the same seed value to the event generator of Monkey, thereby making sure that same input event sequences are generated for both runs of the app. This is to simulate the execution of two copies of the same app with the same user input.

We then ran the birthmark algorithm on the two traces of the same app, thereby computing two birthmarks, and then calculated the similarity coefficient between them. We repeated this procedure for every app in our dataset of 300 apps. The resultant values of similarity coefficient are on the diagonal of figure 3.3. The number of false negatives for results in figure 3.3 is the number of apps which the API birthmark algorithm failed to detect as similar, even though both the traces were generated from the same app. We observed that the number of false negatives dropped to 12 as compared to the experiment in 3.5.5 in which it was 45. This is an expected result, since we are comparing two traces of the same app here as opposed to comparing an app and an obfuscated app shown in figure 3.2.

3.5.7 Detecting Distinct Apps

As much as a birthmark should detect copies of apps, it is important that it should be able to distinguish between two different programs by indicating a low value of similarity between them, thereby proving its credibility. We tested this aspect with our dataset of 300 apps. We did a pairwise comparison for each pair (A, B) where A and B are two apps from our dataset and A is different from B. The results are as shown on non-diagonal points of figure 3.3.

The number of false positives is the number of apps for which the API birthmark algorithm produced similarity coefficient above the threshold during comparison of the app with at least one other app. We observed that the number of false positives increased for this experiment as compared to the experiment described in 3.5.5. It is equal to 115, and this number reduced

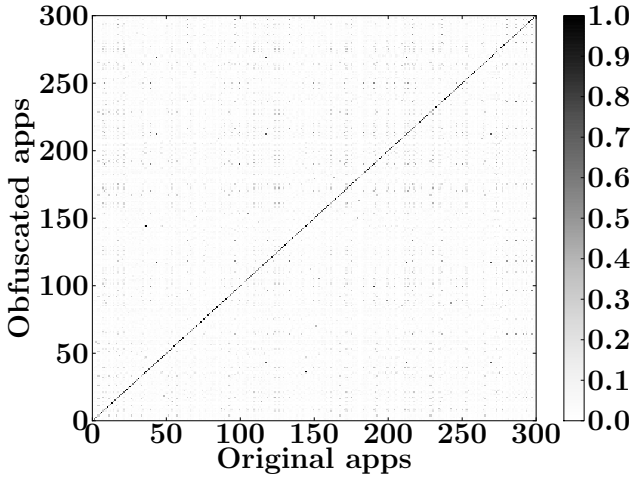

Figure 3.4: API birthmark results on pairwise comparisons for 300 apps (300x300 pairs), keeping the Java API method call in the execution traces. Diagonal shows the result of comparing an app and its obfuscated counterpart. Non-diagonal shows comparisons between app and obfuscated counterpart of a different app.

to 86 after filtering out the different customized versions of the same app. The reasons behind this misclassification are same as those explained earlier in 3.5.5 such as presence of apps developed using a common programming framework and apps using the same advertisement library. Additional filtering techniques are needed to accommodate such apps.

3.5.8 Effect of Inclusion of Java API Methods

During the calculation of API birthmark, we are only observing the Android API method invocations by the app. But an Android app makes use of Java API methods as well. We wanted to evaluate the effect of including this usage of the Java API by the app. Figure 3.4 shows the result of pairwise comparison of 300 apps and 300 obfuscated counterparts of the apps, in which the collected traces have the Java API method calls in addition to the Android API method calls.

Collecting the Java API methods makes the individual traces larger, which in turn adds more items in the API birthmark calculation. As a result, the union of sets of method sequences computed for calculating the Jaccard similarity becomes larger, and hence it results in lower absolute value of similarity coefficient between two apps. For example, the median value of similarity coefficient for an app and obfuscated counterpart of a distinct app is 0.03 without Java and it is 0.04 with Java (the lower this value, the better it is). The median value of similarity

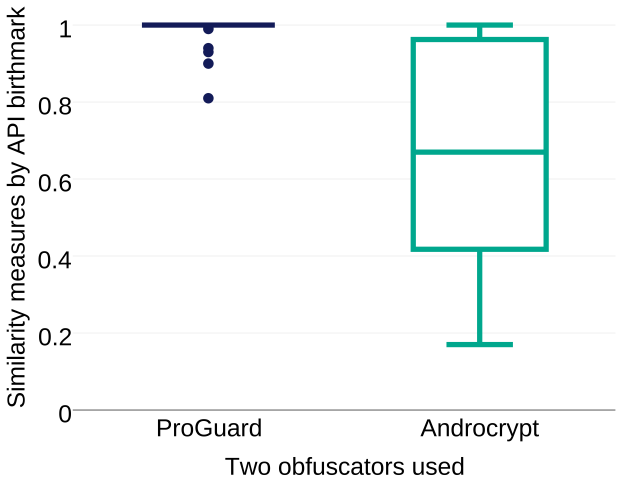

Figure 3.5: Distribution of similarity measures reported by API Birthmark when two different obfuscators were used. Box on left corresponds to using ProGuard as the obfuscator whereas box on right corresponds to use of Androcrypt as obfuscator. The values used to draw the plots are taken from the columns labelled as (PG, B) and (AC, B) in 3.6.

coefficient for an app and obfuscated counterpart of the same app is 0.73 without Java, and the same value is 0.67 with Java.

We therefore concluded that inclusion of Java API methods doesn't have an absolute effect on the birthmark computation. We decided not to include the Java API methods in our birthmark computation, since it results in lower values similarity coefficient.

3.5.9 Experiments with ProGuard as Obfuscator

In our experiments so far, we used Androcrypt as the obfuscator to produce the obfuscated counterparts of original apps. As shown in §3.2, Androcrypt is one of the strongest obfuscator that we know of and hence produces strong obfuscated code. The stronger the obfuscation, the lower the similarity measures reported by any similarity detector. As a corollary, the weaker the obfuscation, the higher the similarity measures. Hence, if we use any other obfuscator which is weaker than Androcrypt, the scores reported by our API birthmark technique should only get higher. To confirm this hypothesis, we performed the following experiment. We took the same set of 53 apps shown in the 3.6, and used off-the-shelf obfuscator – ProGuard – to obfuscate the apps. We ran API birthmark technique on the pair of apps (A, $A_{obfuscated}$) where A is an original app and $A_{obfuscated}$ is same app obfuscated using ProGuard. We then ran the API birthmark on each pair of apps. The results are shown in the column labelled as (PG, B) in 3.6.

Figure 3.6: Similarity scores reported by two app similarity detectors, Androguard and API Birthmark, when app and its obfuscated version are given as input. The first column lists the category of the app. The second column gives the app package name. The last four columns show results for different combinations of obfuscators and similarity detectors. The abbreviations in the four columns stand for the following:
(PG, AG): ProGuard as obfuscator, Androguard as similarity detector.
(AC, AG): Androcrypt as obfuscator, Androguard as similarity detector.
(PG, B): ProGuard as obfuscator, API Birthmark as similarity detector.
(AC, B): Androcrypt as obfuscator, API Birthmark as similarity detector.

Category	App/Package name	App size (in KB)	(PG, AG)	(AC, AG)	(PG, B)	(AC, B)
Development	alogcat	40	0.76	0.03	1.00	0.68
Games	chessclock	92	0.78	0.09	1.00	0.74
Games	tictactoe	1,963	0.24	0.01	0.93	0.57
Games	solitaire	70	0.84	0.01	1.00	1.00
Games	sokoban	109	0.82	0.07	1.00	0.88
Games	kaesekaestchen	148	0.79	0.05	1.00	0.69
Games	atomix	210	0.90	0.02	1.00	0.23
Games	lexic	245	0.68	0.02	0.81	0.26
Games	androc	539	0.79	0.01	1.00	0.17
Games	games.memory	2,080	0.68	0.03	0.90	0.25
Games	bomber	455	0.80	0.11	1.00	0.67
Games	blockinger	801	0.48	0.01	1.00	0.17
Games	blokish	421	0.71	0.02	1.00	1.00
Games	amazed	15	0.44	0.10	1.00	0.67
Games	opensudoku	211	0.95	0.01	1.00	0.75
Internet	blitzmail	280	0.22	0.00	1.00	1.00
Internet	connectbot	858	0.72	0.00	1.00	0.70
Internet	reddit	759	0.47	0.00	1.00	0.50
Multimedia	binauralbeat	965	0.63	0.02	1.00	0.41
Multimedia	avs234	162	0.79	0.02	1.00	1.00
Multimedia	zooborns	39	0.75	0.05	1.00	0.41
Multimedia	zxing	720	0.61	0.00	1.00	0.95
Multimedia	photostream	134	0.77	0.03	0.99	0.61
Navigation	pedometer	46	0.97	0.03	1.00	0.39
Navigation	stardroid	2,188	0.53	0.51	1.00	0.96
Office	aarddict	1,852	0.29	0.00	1.00	0.65
Office	babycaretimer	457	0.23	0.01	1.00	0.46
Office	calculator	77	0.66	0.01	0.99	0.56
Office	coinflip	422	0.67	0.04	0.99	0.97
Office	birthdroid	79	0.78	0.08	1.00	0.85
Office	Keyer	82	0.65	0.04	1.00	0.40
Office	TeaTimer	216	0.77	0.04	1.00	0.51
Office	aGrep	54	0.88	0.04	1.00	1.00
Office	simplydo	64	0.73	0.03	1.00	0.56
Office	tipitaka	502	0.49	0.00	0.99	0.77
Office	mileage	366	0.72	0.56	1.00	0.96
Office	wikinotes	119	0.92	0.28	1.00	0.31
SMS	autoanswer	88	0.54	0.14	1.00	1.00
SMS	autoawayy	112	0.69	0.03	1.00	1.00
Reading	adsdroid	116	0.65	0.01	1.00	1.00
Reading	andquote	38	0.88	0.06	1.00	0.24
Education	antikythera	531	0.71	0.08	1.00	0.83
Education	angulo	18	0.93	0.08	1.00	1.00
System	airpushdetector	33	0.86	0.13	1.00	0.65
System	autostarts	288	0.43	0.00	0.90	0.42
System	appalarm.pro	121	0.80	0.01	1.00	0.22
System	apptracker	140	0.57	0.02	0.94	0.46
System	asqlitemanager	339	0.71	0.01	1.00	0.23
System	batterydog	21	0.73	0.38	1.00	1.00
System	httpmon	74	0.80	0.01	1.00	1.00
System	adbWireless	378	0.36	0.00	1.00	0.92
System	androsens	21	0.91	0.21	1.00	0.54

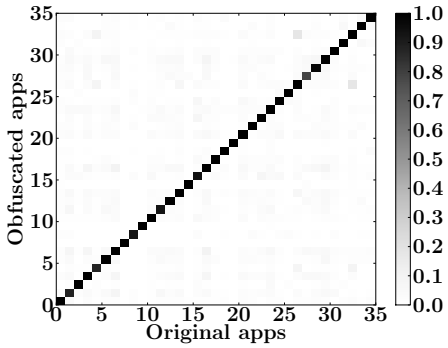

Figure 3.7: API birthmark results on pairwise comparisons for 35 apps (35x35 pairs) with traces generated manually. Diagonal shows the result of comparing an app and its obfuscated counterpart. Non-diagonal shows comparisons between app and obfuscated counterpart of a different app.

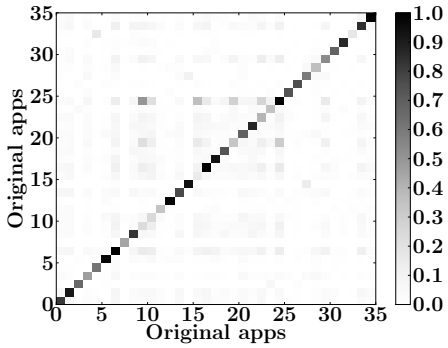

Figure 3.8: API birthmark results on pairwise comparisons for 35 apps (35x35 pairs) with traces generated automatically using Monkey, keeping the Java API method call in the execution traces. Diagonal shows the result of comparing an app and its obfuscated counterpart. Non-diagonal shows comparisons between app and obfuscated counterpart of a different app.

For comparison, we also give values reported by API birthmark when Androcrypt was used instead of ProGuard in the column labelled as (AC, B) in 3.6. We show box plot of the values in Figure 3.5. As the plot shows, when an off-the-shelf obfuscator such as ProGuard is used, API birthmark performs at least as good as or better, compared to the case when Androcrypt is used.

3.5.10 Automated Execution using Monkey

The API birthmark is a dynamic birthmark that requires run-time trace generated during execution of the app. Since Android apps are interactive in nature, you need to provide user inputs such as clicks, gestures and touch events to them on a continuous basis during their run. During

manual execution of the app, the user of the app provides these inputs. But manual execution is time consuming and limits the number of apps that one can experiment with. Therefore, we wanted to automate the execution of Android apps.

In order to run the apps automatically, you need a tool that will generate events automatically as opposed to manually supplying the events. We found that the Monkey [23] program shipped with Android SDK is one of the best tools available for such purposes. The primary purpose of Monkey is to stress-test the app by simulating the generation of various input events such as clicks and touches. But by using this automatic generation of input events, you can automate the execution of an Android app. Monkey can be configured to run with a number of command line options. For our setup, we used the following options: the seed value of random number generator, total number of input events to be generated and option to ignore the exceptions during app's execution. For a given app and a seed value, monkey generates a particular but random sequence of events. We give the same seed value for execution of both, the original app and the plagiarized app, so that the exact same operations are exercised while executing the pair of apps using monkey.

Using Monkey, we succeeded in automatic execution and trace collection for 350 apps. Use of Monkey for more apps in our dataset failed due to a number of reasons. The events supplied by Monkey may not be context-sensitive *i.e.,*they may not be relevant for the current execution state of the app, resulting into the app's crash, thereby producing no trace. Monkey does not give any preference to frequently occurring events over infrequent ones during the event generation. Moreover, it is hard to predict the right fraction of UI events and system events needed to run the app exhaustively, that can be given as input to Monkey. This results in poor coverage of the app's functionality, thereby leading to poor quality of the generated traces. The incomplete coverage of the app's functionality in traces directly affects the computation of the API birthmark. We conducted an experiment to study the impact of using Monkey for the collection of execution traces of Android apps. We chose 35 apps randomly from our dataset. In addition to the traces that were generated earlier using Monkey for these apps, we manually executed these apps by interacting with the app as a user and collected another set of traces. Figure 3.7 and 3.8 shows the results of pairwise comparisons of these apps and their obfuscated counterparts, with manually generated traces and traces generated using Monkey,

respectively. It is evident from the two graphs that the API birthmark algorithm gives more accurate similarity coefficients when the traces have been generated manually. We therefore believe that use of a more robust tool for automated execution of Android apps would give better results. In future, we plan to repeat our experiments using other tools available such as Dynodroid [63] which has better coverage of the app's functionality than Monkey. However, there are trade-offs of choosing each of these tools *e.g.,*Dynodroid suffers from a performance penalty (Dynodroid is 5X slower than Monkey) and hence may not be suitable for app execution at a large scale.

3.5.11 Attacks and Limitations

Let us now look at the possible attacks on the API birthmark. An attacker can inject random API method calls in the source code of the app and thus skew the API birthmark. Such API method call injection may be done by employing techniques used in the creation of polymorphic viruses [92]. For this attack to be effective, each newly added method call or the method calls added as a group should produce zero side effects. Inferring the dummy methods from the API *automatically* is a hard problem. The attacker will have to resort to manual techniques for finding such API methods or method sequences. Indeed, if discovery of such methods or method sequences is an easy task, we can generate them ourselves, and filter them out from the dynamic traces, before running the API birthmark algorithm. Moreover, the attacker will have to insert at least 30 percent new sequences (since similarity threshold is 0.3), which leads to increased code size and additional runtime cost. In general, the attacks on the API birthmark have cost overhead and involve manual work which offer less incentives for the attacker to implement them.

First limitation of our approach is its scalability. In Section 3.5.10, we explain the difficulties faced in automating the execution of Android apps. Even the best tool among all of the available ones, namely *monkey* ([23]), does not yield useful traces for a number of available apps because of many reasons such as crashes in the middle of the execution or generation of extremely short traces. Also, the encrypting obfuscator works only partially on apps that contain certain classes such as *ContentProvider*, as described in section 3.2. These factors put a limit on the number of apps with which we can experiment. Unless the automatic execution of

Android apps becomes pratical for large datasets, experimenting on them remains a subject of future work. One practical solution could be to run the static analyses first on the larger dataset of apps which would narrow down the suspected plagiarized apps. We can then run the API birthmark algorithm on the smaller dataset.

Our setup includes automatic execution of apps by using the monkey program shipped with Android SDK. The fact that we rely on monkey for generating input events has two implications.

1. Monkey generates a random sequence of input events corresponding to a seed value. Greater the coverage of input events provided by monkey, better it is to compute similarity using API birthmark algorithm. To increase the runtime test coverage, better tools are needed for automatic testing of Android apps.

2. For a given seed value, monkey generates a stream of events such as clicks, touches or gestures to stress test the app. We give the same seed value to both, the original and the plagiarized app, so that the same sequence of events is generated while executing them. However, if there are small UI tweaks in the plagiarized app, some of the generated events would be impossible to operate on the plagiarized app. For example, if monkey has generated an event of type *Send touch-event-at-(x,y)=(1.0, 4.0)*, but the plagiarized app has moved the button to a different location, then an attempt to execute this operation may lead to unexpected consequences. Event generation in such cases may be accomplished by coupling monkey with other powerful GUI automation approaches such as Sikuli Script [90] which provides a mechanism to programmatically control GUI elements in the automation scripts using their screenshots. Using Sikuli Scripts, one can use screenshots of GUI elements in the testing tool rather using raw coordinates, thus eliminating the scenarios where there are minor modifications in the GUI.

3.6 Summary

Code plagiarism is an important problem that plagues the mobile app development community, and serves as a popular vehicle for the delivery of malicious apps. Although the community has developed a number of code similarity metrics to combat plagiarism, they rely on syntactic

features of the code to operate.

In this chapter, we show that such syntactic similarity measures are broken, because they can easily be evaded using simple obfuscations. We develop a robust API birthmark-based approach to detect code similarity. Our experiments on a dataset of Android apps shows that the birthmark-based approach is effective at detecting code plagiarism even with obfuscated apps.

Chapter 4

Related Work

Birthmark-based software theft detection. The technique of constructing *software birthmarks* was proposed earlier by other researchers in the context of traditional desktop programs. To our knowledge, we are the first to use software birthmarks on mobile applications. Software birthmarks can be categorized into two classes, dynamic and static. Myles and Collberg introduced the whole-program-path dynamic birthmark [68]. Our approach of using dynamic API birthmarks has been built upon the techniques proposed in [83] and [80]. In [83], Tamada *et al.* proposed a dynamic API birthmark based on observations of the interaction of windows application with its environment. Schuler *et al.* [80] put forward an improved version of dynamic API birthmark that is based on watching the program interaction at the level of objects which results in shorter API sequences. These two projects were done in the context of traditional desktop software (Windows applications and Java programs, respectively). We have applied those ideas to the domain of mobile apps, particularly for the problem of identifying plagiarized mobile apps.

A slightly different problem of assessment of similarity between two algorithms was targeted in [94] where they used two dynamic value-based approaches, namely N-version programming and annotation.

Among different bodies of work that use static birthmarks, GP_{LAG} [60] is a tool to detect plagiarism in software by mining program dependence graphs. A birthmark for Java applications was developed by Lim *et al.* in [59] that identifies and uses possible stack patterns that may be formed during program execution by analyzing the Java bytecode statically. Use of opcode-level k-grams as software birthmarks was done in [67] by Myles and Collberg.

Code clone detection The problem of detecting clones in software code has been well studied. A survey paper by Bellon *et al.* [78] gives a good comparison and evaluation of various

clone detection tools for traditional software programs. Various techniques have been explored to detect code clones that derive and use different type of information from the code such as text and tokens [91], metric vectors [54], abstract syntax trees [30] and program dependency graphs [56].

Detecting plagiarized mobile apps. The problem of identifying plagiarism in mobile apps has attracted a lot of attention recently. In [98], the idea was to generate a unique hash of the app from its Dalvik bytecode by using a fuzzy hashing technique. In [97], the authors use a number of features of the application such as the android API methods, permissions requested by the application etc. to construct a feature vector and then employ Jaccard distance to define distance between two feature vectors. In [35], the approach is to first group together similar apps based on certain features and then apply program-dependence-graph-based techniques to detect cloning.

To summarize, the proposed approaches employ different techniques such as fuzzy hashing [98], feature hashing [47], program dependence graph (PDG) [35], [50] and module decoupling [97] for computing unique fingerprint of the app. These techniques rely on extracting static properties of the application by analyzing the application's source code. These can be defeated easily by code obfuscations. We are proposing an effective plagiarism detection technique for mobile apps based on dynamic analysis which is resilient to code obfuscations.

Mining API mappings. An upcoming survey article by Robillard *et al.* [77, §6] provides a good overview of prior work on mining API mappings. Among these, the work most directly related to ours is the MAM project [95]. MAM's goal is the same as Rosetta's, *i.e.,* to mine software repositories to infer how a source API maps to a target API. The MAM prototype was targeted towards Java as the source API and C# as the target API. Despite sharing the same goal, MAM and Rosetta differ significantly in the approaches that they use, each with its advantages and drawbacks. To mine API mappings between a source and a target API, MAM relies on the existence of software packages that have been ported manually from the source to the target platform. For each such software package, MAM then uses static analysis and name similarity to "align" methods and classes in the source platform implementation with those of the target platform implementation. Aligned methods are assumed to implement the same functionality.

MAM's use of static analysis allows it to infer a large number of API mappings (about

25,800 mappings between Java and C#). It also allows MAM to infer likely mappings between arguments to methods in the source and target APIs, which Rosetta does not currently do. However, unlike Rosetta, MAM requires that the same source application be available on the source and target platforms. MAM's approach of aligning classes and methods across two implementations of a software package does not allow the inference of likely API mappings if there are similar, but independently-developed applications for the source and target platforms. MAM's approach is also limited in that it uses name similarity as the only heuristic to bootstrap its API mapping algorithm. In contrast, Rosetta uses a number of attributes combined together as factors, and can easily be extended to accommodate new similarity attributes as they are designed. Most importantly, while MAM uses a purely *syntactic* approach to discover likely API mappings, Rosetta's approach uses similarities in application *behavior*.

Leveraging API mappings. Nita and Notkin [71] develop techniques to allow developers to adapt their applications to alternative APIs. They provide a way for developers to specify a mapping between a source and a target API, following which a source to source transformation automatically completes the transformations necessary to adapt the application to the target API. Rosetta can potentially complement this work by inferring likely mappings.

GUI APIs. Androider [81] is a tool to reverse-engineer application GUIs. Androider uses aspect-oriented programming techniques to extract a platform-independent GUI model, which can then be used to port GUIs across different platforms. While Androider provides a GUI for a target platform by analyzing GUIs of a source platform at runtime, Rosetta instead infers API mappings, and is not restricted to GUI-related APIs. Bartolomei *et al.* [29] analyzed wrappers between two Java GUI APIs and extracted common design patterns used by wrapper developers. They focused on mapping object types and identified the challenges faced by wrapper developers. Method mappings given by Rosetta can possibly be used along with their design patterns to ease the job of writing wrappers.

API aids and learning resources. A number of prior projects provide tool support to assist programmers working with large, evolving APIs (*e.g.,* [36, 38, 64, 79, 88, 93, 96]). The programmer's time is often spent in determining which API method to use to accomplish a particular task. These projects use myriad techniques to develop programming assistants that ease the task of working with complex APIs. Rosetta is complementary to these efforts, in that

it works with *cross-platform APIs*.

A related line of research is on resources for API learning (*e.g.,* [32, 37, 48, 74, 84]). These projects attempt to ease the task of a programmer by synthesizing API usage examples, evolution of API usage, and extracting knowledge from API documentation. Again, most of these techniques work on APIs on a single platform. Rosetta's *cross-platform* approach can possibly be used in conjunction with these techniques to facilitate cross-platform API learning.

Factor graphs. Finally, Rosetta's probabilistic inference approach was inspired by other uses of factor graphs in the software engineering literature. Merlin [61] uses factor graphs to classify methods in Web applications as sources, sinks and sanitizers. Such specifications are useful for verification tools, which attempt to determine whether there is a path from a source to a sink that does not traverse through a sanitizer. To infer such specifications, Merlin casts a number of heuristics as factors, and uses belief propagation. Kremenek *et al.* [55] also made similar use of factor graphs to infer specifications of allocator and deallocator methods in systems code. Factor graphs are just one approach to mining specifications; a number of prior projects have considered other techniques for data mining and inferring belief for specification inference and bug detection (*e.g.,* [40, 58, 62]). Future work could consider the use of these inference techniques in Rosetta as well.

Using NLP techniques in software engineering research. Recently, researchers have succcessfully applied techniques from natural language processing (NLP) field to some of the software engineering tasks. In [49], authors argue that code found in software programs follows repetitive patterns, similar to the natural language text and they used n-gram language models to predict the likely next token to occur after a given sequence of tokens. Nguyen *et al.* proposed a semantic language model ([70]) that extends the n-gram language model and provides improved accuracy for code token prediction.

Haghighi *et al.* [46] proposed a solution to infer translation dictionary between two languages by using non-parallel text of the two languages. We use their method as our inference engine and to our knowledge, we are the first to apply NLP technique for the problem of inferring mappings between APIs. [42] and [53] are some of the older papers in NLP domain that induce translation dictionary between non-parallel text.

Chapter 5

Conclusion

The work described in this dissertation uses the interaction of mobile apps with the underlying platform as an abstract representation for the high level behavior of the apps. Our work demonstrates that this representation in the form of sequences of API methods invoked by the mobile apps can be used as as a resource in detecting similarities between mobile apps or similarities between mobile platform APIs. The systems we have developed can be used by mobile app developers for cross-platform app development or by platform vendors in keeping the app repositories free from plagiarized apps. These systems are one small piece in the bigger puzzle of automating the tedious and time-consuming tasks in mobile application development and maintenance. We now describe a few research questions in this area.

5.1 Future Directions

App development on multiple devices and platforms. Mobile app developers face many challenges in the process of developing apps. Cross-platform app development is only one of the problems. Another problem is the existence of a range of devices on a single platform, each one having a different screen size and resolution. App developers have to ensure that their apps show nicely and correctly for each of these devices available on the same platform. Development tools that shield the programmers from some of the device-specific considerations would be of great help.

A number of different computing platforms are emerging and would be deployed in the near future. Car manufacturers have started offering either in-built apps in cars [7] or apps on other wearable devices that work with connectivity systems in cars [6]. Apple, Samsung and other vendors have introduced smart-watches loaded with apps. With Internet-of-Things, a plethora of other devices will soon have apps installed on them. It would be interesting to see how apps

are built for each of these computing platforms. As developers try to develop the same app for every single such device, helper tools would be needed to make this development process more manageable.

Productivity tools for programmers. An interesting fact about the ecosystem of mobile apps is the huge scale at which it is growing. The two most popular platform vendors, Apple and Google, each have over 1.4 million apps in their app stores as of May 2015. [18]. These app stores attracted over 250,000 registered developers each in 2014 [4]. Thus, mobile apps are being developed and deployed at a large scale by a growing community of developers. The availability of mobile apps in huge numbers can be leveraged by applying statistical techniques on these big codebases to develop new insights that would help in development of new mobile apps. Systems such as recommendation engines for suggesting different pieces of the code at different granularity levels and semi auto-completion systems that help fill in partial code templates can be built in this manner to help app developers.

References

[1] A tool for reverse engineering Android apk files. https://code.google.com/p/android-apktool/.

[2] Allatori Java obfuscator. http://www.allatori.com/.

[3] Androguard. http://code.google.com/p/androguard/wiki/Usage#Androsim.

[4] App stores growth accelerates in 2014. http://blog.appfigures.com/app-stores-growth-accelerates-in-2014/.

[5] AppleScript scripting language. http://tinyurl.com/k38uroq.

[6] BMW, Porsche roll out apple watch apps for cars. http://www.usatoday.com/story/money/cars/2015/04/24/bmw-porsche-apple-watch/26307085/.

[7] Cars are the new smartphones. http://www.theverge.com/2014/1/5/5276536/cars-are-the-new-smartphones-chevrolet-adding-lte-and-app-store-to-2015-models.

[8] Dalvik Debug Monitor Server (DDMS). http://developer.android.com/tools/debugging/ddms.html.

[9] Dalvik Debug Monitor Server (DDMS). http://docs.eoeandroid.com/tools/debugging/debugging-tracing.html.

[10] Dare: Dalvik retargeting. http://siis.cse.psu.edu/dare/.

[11] DashO Java obfuscator. http://www.preemptive.com/products/dasho/overview.

[12] FOSS apps for Android. https://f-droid.org.

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

[14] iTunes app store passes 1.5M apps, 100B downloads, 30B paid to developers to date. http://techcrunch.com/2015/06/08/itunes-app-store-passes-1-5m-apps-100b-downloads-30b-paid-to-developers/.

[15] Jaccard Index. http://en.wikipedia.org/wiki/Jaccard_index.

[16] MobileFirst. http://www-03.ibm.com/software/products/en/mobilefirstfoundation.

[17] NetworkX graph library. https://networkx.github.io/.

[18] Number of apps available in leading app stores as of May 2015. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[19] PhoneGap. http://phonegap.com/.

[20] ProGuard. http://proguard.sourceforge.net/.

[21] Sencha. https://www.sencha.com/.

[22] Soot: A Java optimization framework. http://www.sable.mcgill.ca/soot/.

[23] UI/Application exerciser monkey. http://developer.android.com/tools/help/monkey.html.

[24] Who is winning the game of devices? http://www.greenbookblog.org/2014/09/17/who-is-winning-the-game-of-devices-mobile-vs-desktop/.

[25] Assembla. J2ME Android Bridge. http://www.assembla.com/spaces/j2ab/wiki.

[26] Francis R. Bach and Michael I. Jordan. A probabilistic interpretation of canonical correlation analysis. Technical report, University of California, Berkeley, 2005.

[27] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, 1995.

[28] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, 1995.

[29] T.T. Bartolomei, K. Czarnecki, and R. Laandmmel. Swing to SWT and back: Patterns for API migration by wrapping. In *Proceedings of the 26th International Conference on Software Maintenance*, 2010.

[30] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, 1998.

[31] I. D. Baxter, C. Pidgeon, and M. Mehlich. Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[32] R. Buse and W. Weimer. Synthesizing API usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[33] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, 2004.

[34] OW2 Consortium. ASM toolkit. http://asm.ow2.org.

[35] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *17th European Symposium on Research in Computer Security*, 2012.

[36] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. *ACM Trans. Softw. Eng. Methodol.*, 20(4), 2011.

[37] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[38] D. Dig, C. Comertoglu, D. Marinov, R. Johnson, and D. Thomas. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, 2006.

[39] Stphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance*, 18(1), 2006.

[40] D. Engler *et al*. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, 2001.

[41] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, 2011.

[42] Pascale Fung. Compiling bilingual lexicon entries from a non-parallel english-chinese corpus. In *Proceedings of the Third Workshop on Very Large Corpora*, 1995.

[43] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: examining the landscape and impact of Android application plagiarism. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, 2013.

[44] Amruta Gokhale, Daeyoung Kim, and Vinod Ganapathy. Data-driven inference of API mappings. In *2nd Workshop on Programming for Mobile and Touch*, 2014.

[45] Google. Android API reference. `http://developer.android.com/reference/packages.html`.

[46] Aria Haghighi, Percy Liang, Taylor Berg-kirkpatrick, and Dan Klein. Learning bilingual lexicons from monolingual corpora. In *Proceedings of ACL-08: HLT*, 2008.

[47] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: a scalable system for detecting code reuse among android applications. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, 2012.

[48] S. Hens, M. Monperrus, and M. Mezini. Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[49] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[50] Jonathan Crussell and Clint Gibler and Hao Chen. AnDarwin: scalable detection of semantically similar Android applications. In *18th European Symposium on Research in Computer Security*, ESORICS '13, 2013.

[51] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement,*, 2013.

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

[53] Philipp Koehn and Kevin Knight. Learning a translation lexicon from monolingual corpora. In *Proceedings of the ACL-02 Workshop on Unsupervised Lexical Acquisition - Volume 9*, ULA '02, 2002.

[54] Kostas Kontogiannis and Renato de Mori and Morris Bernstein and M. Galler and Ettore Merlo. Pattern matching for design concept localization. In *Proc. Working Conf. Reverse Engineering (WCRE)*, 1995.

[55] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[56] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, 2001.

[57] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theory*, 47(2), 2001.

[58] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, 2005.

[59] Hyun-il Lim, Heewan Park, Seokwoo Choi, and Taisook Han. Detecting theft of java applications via a static birthmark based on weighted stack patterns. 2008.

[60] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, 2006.

[61] B. Livshits, A. Nori, S. Rajamani, and A. Banerjee. Merlin: Inferring specifications for explicit information flow problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[62] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, 2005.

[63] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

[64] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[65] MicroEmulator. `http://www.microemu.org/`.

[66] K. Murphy. Bayes Net Toolbox for Matlab, October 2007. `http://code.google.com/p/bnt/`.

[67] Ginger Myles and Christian Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, 2005.

[68] Ginger Myles and Christian S. Collberg. Detecting software theft via whole program path birthmarks. In *7th Information Security Conference*, 2004.

[69] Netmite Corporation. App Runner. `http://www.netmite.com/android/`.

[70] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

[71] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd International Conference on Software Engineering*, May 2010.

[72] Oracle. Java ME API reference. `http://docs.oracle.com/javame/config/cldc/ref-impl/midp2.0/jsr118/index.html`.

[73] Oracle. Sun Java wireless toolkit for CLDC, 2.5.1. `http://java.sun.com/products/sjwtoolkit/download-2_5_1.html`.

[74] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. M. Paradkar. Inferring method specifications from natural language API descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[75] python-levenshtein-0.10.2. `pypi.python.org/pypi/python-Levenshtein`.

[76] Qt API mapping for iOS developers. `http://www.developer.nokia.com/Develop/Porting/API_Mapping`.

[77] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 2013.

[78] S Bellon and R Koschke and G Antoniol and J Krinke and E Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*.

[79] T. Schafer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th International Conference on Software Engineering*, 2008.

[80] David Schuler, Valentin Dallmeier, and Christian Lindig. A dynamic birthmark for java. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, 2007.

[81] E. Shah and E. Tilevich. Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms. In *Workshop on Next-generation Applications of Smartphones*, 2011.

[82] Mark D. Syer. Empirical studies of mobile apps and their dependence on mobile platforms. Master's thesis, Queen's University, 2013.

[83] Haruaki Tamada, Keiji Okamoto, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Dynamic software birthmarks to detect the theft of windows applications. In *International Symposium on Future Software Technology*, 2004.

[84] G. Uddin, B. Dagenais, and M. P. Robillard. Temporal analysis of API usage concepts. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[85] Vaibhav Rastogi and Yan Chen and Xuxian Jiang. DroidChameleon: evaluating Android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, 2013.

[86] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, 1981.

[87] Windows phone interoperability: Windows phone API mapping. `http://windowsphone.interoperabilitybridges.com/porting`.

[88] Z. Xing and E. Stroulia. API-evolution support with DiffCatchUp. *IEEE Transactions on Software Engineering*, 33(12), 2007.

[89] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 2003.

[90] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: Using GUI screenshots for search and automation. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, 2009.

[91] Yoshiki Higo and Yasushi Ueda and Toshihiro Kamiya and Shinji Kusumoto and Katsuro Inoue. On software maintenance process improvement based on code clone analysis. In *Proceedings of the 4th International Conference on Product Focused Software Process Improvement*, PROFES '02, 2002.

[92] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, BWCCA '10, 2010.

[93] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, and J. Zhao. Automatic parameter recommendation for practical API usage. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[94] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, 2012.

[95] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of the 32nd International Conference on Software Engineering*, 2010.

[96] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: mining and recommending API usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, 2009.

[97] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, 2013.

[98] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, 2012.

[99] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, 2012.