Studying and Enabling Reuse in Android Mobile Apps

Andrew Steven Holtzhauer

Stafford, VA

Bachelor of Science, College of William and Mary, 2012

A Thesis presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Master of Science

Department of Computer Science

The College of William and Mary
August 2014

APPROVAL PAGE

This Thesis is submitted in partial fulfillment of
the requirements for the degree of

Master of Science

Andrew Steven Holtzhauer

Approved by the Committee, August 2014

Committee Chair
Assistant Professor Denys Poshyvanyk, Computer Science
The College of William and Mary

Associate Professor Peter Kemper, Computer Science
The College of William and Mary

Assistant Professor Collin McMillan, Computer Science
University of Notre Dame

ABSTRACT

In the recent years, studies of design and programming practices in mobile development are gaining more attention from researchers. Several such empirical studies used Android applications (paid, free, and open source) to analyze factors such as size, quality, dependencies, reuse, and cloning. Most of the studies use executable files of the apps (APK files), instead of source code because of availability issues (most of free apps available at the Android official market are not open-source, but still can be downloaded and analyzed in APK format). However, using only APK files in empirical studies comes with some threats to the validity of the results. In this paper, we analyze some of these pertinent threats. In particular, we analyzed the impact of third-party libraries and code obfuscation practices on estimating the amount of reuse by class cloning in Android apps. When including and excluding third-party libraries from the analysis, we found statistically significant differences in the amount of class cloning 24,379 free Android apps. Also, we found some evidence that obfuscation is responsible for increasing a number of false positives when detecting class clones. Finally, based on our findings, we provide a list of actionable guidelines for mining and analyzing large repositories of Android applications and minimizing these threats to validity.

While in our initial work we studied different factors that impact reuse in Android apps, we also designed and implemented an approach to help facilitate the enabling of reuse in Android mobile applications. Although mobile app stores may have a list of similar apps to present to the user, this list may not be complete and/or accurate. Detecting similar applications is a notoriously difficult problem, since it implies that similar highlevel requirements and their low-level implementations can be detected and matched automatically for different applications. We designed an approach for automatically detecting Closely reLated applications in ANdroid (CLANdroid), which helps detect similar Android applications based on a given Android mobile app. CLANdroid is an extension to a novel approach by CLAN, which is a previously published approach that is included in this thesis for completeness purposes. Our main contributions are an extension to a framework of relevance and a novel algorithm that computes a similarity index between Java and Android applications using the notion of semantic layers that correspond to packages and class hierarchies. To evaluate CLANdroid we extracted a goldset for each of the 14,450 apps in our dataset, which consisted of apps that were deemed similar based on the app's page on Google Play. We compared five different ranking methods: API calls, identifiers, intents, permissions, and phone sensors. The results show that when considering the whole dataset, the identifiers ranking method is most effective.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

My thanks go to my advisor, without whom this would not be possible. Dr. Denys has always been supportive, and his tutelage for the past two years has helped me to grow not only as a student, but as a person more than I have in the past decade. I could not ask for a better leader, and will be forever grateful for his kindness and teachings.

To my sister. The one person who somehow always knew when to call and interrupt me when I was the most busy, and who never thought twice about spamming me with nonsensical text messages. Nonetheless, I love you so incredibly much, and even if we can't stand being around each other for more than a week at a time, thank you for always being there for me, and I hope you know that I'm always here for you - no matter what.

To Mario and Chris. I cannot even begin to imagine where I would be or how I would have handled the challenges of both life and school without either of you. I will never forget the memories and bonds we built together, whether it be in the lab or at Sweet Frog. You both have helped me to understand the meaning of what a wise man once said: "The more you do, the more expert you are." There is only one thing I have left to say to the two of you: Vamos! A la! Playa!!

To my mother and father, who have always supported me and provided me with more love than I could ask for

LIST OF TABLES

LIST OF FIGURES

Studying and Enabling Reuse in Android Mobile Apps

Chapter 1

Revisiting Android Reuse Studies

Developing mobile applications differs from desktop and web applications in several dimensions. It is not only about the revenue models or the size of the applications, but also about programming practices, hierarchy and structure of development teams, as well as other factors. For example, testing mobile applications highly benefits from crowdsource-based approaches that assume testing newly released applications (or updates) on different devices, operating systems, and under different connection modes (e.g., offline, WiFi, cellular network). Also, reuse in mobile applications (hereinafter referred as apps) is ongoing and widespread, in particular because the apps are highly dependent on the APIs [56, 57, 72] and the distribution model in markets allows developers to sell the same apps several times by (re)packaging them with different GUI elements.

As of today, only a handful of papers have analyzed mobile apps and their ecosystems to understand the factors that distinguish mobile apps and their development processes from desktop and web applications [12, 26, 36, 53, 56, 57, 72, 73]. For example, Minelli and Lanza [53] and Syer *et al.* [73] suggest that practices for desktop and server-based applications may not necessarily apply to mobile apps. Most of these related papers used a similar approach that consists of analyzing the code or metadata available in public markets. In the particular case of Android, APK (Application PacKage) files have been analyzed. It should be noted that these studies use executable files of the apps (APK files), instead of source code because of availability issues -- most of the free apps available at the Android official market are not open-source, but can still be downloaded and analyzed in APK format. During the analysis, the APK files are converted to JAR files or decompiled to Java source code. *However, the building and packaging model of Android apps (APK files) may introduce some threats to validity of the results of empirical studies.* Even in the case of open source apps, according to [53, 72] some apps include the source code of third-party libraries.

As described in the Android developer guide [16], JAR libraries referenced by the source code of Android apps are imported into APK files at build time. Therefore, when the Android build system converts the .class files into a DEX file, a converter tool is called to extract .class files from JAR libraries and consider them as local .class files compiled from the application source code. Consequently, when converting APK files to JAR files or to Java source code, all the files are under the same root directory (app classes and third-party libraries), thus following the Java rules for organizing files under packages. In addition, obfuscation is a common practice recommended in the Android developer guide [18] to protect security protocols and other application components from reverse engineering attacks. Also, obfuscation is used to hide illegal reuse and avoid licensing issues [69].

Including the code of third-party libraries in the APK files and ignoring obfuscation practices are threats to validity of empirical studies using APK files, in particular the ones aimed at analyzing class cloning/reuse in Android apps. For example, because of the build process, it is not possible to distinguish directly between code referenced as a library and code that was copied and modified from other applications or third-party libraries. Also, signature-based techniques for detecting class cloning, such as Software Bertilonage [9, 8], are sensitive to obfuscation, mainly to transformations such as renaming, ordering (e.g., changing order or methods, or changing order of parameters in methods), and aggregations (e.g., inline and outline methods, cloning methods) [6]. In general, the study by Schulze and Meyer showed that obfuscation by renaming identifiers reduces the effectiveness of text-based clone detectors [69].

Previous studies have not considered the impact of obfuscated code and third-party libraries on the measurements of class cloning in Android apps. Only a recent study by Mojica *et al.* [56] removed obfuscated classes from their dataset when computing the amount of class cloning on Android apps. Therefore, in this thesis we provide empirical evidence on how third-party libraries and obfuscated code can impact reuse measurements. We computed the amount of classes reused on a large set of 24,379 free apps downloaded from Google play, including/excluding third-party libraries, and including/excluding obfuscated apps (that we detected using our algorithms). For detecting clones we used a signature-based approach as in [56, 57]. For detecting apps with obfuscated code we used a simple heuristic we defined after manually inspecting a large sample of obfuscated apps in our dataset.

The results of this study show that there are significant and large differences, in terms of statistical significance and effect size, between the amount of class signatures reused in Android apps when including and excluding third-party libraries. Moreover, although the impact of obfuscated code is negligible when detecting cloned classes in Android apps, we found evidence of false positives declared as clones by the signature-based approach. Therefore, researchers analyzing/mining APK files should consider carefully when to include/exclude third-party libraries and obfuscated code, in particular for studies that use lexical information extracted from the files (i.e., identifiers) and signatures, or studies aimed at measuring similarities among apps.

1.1 Related Work

Several recent papers have analyzed software evolution- and maintenance-related aspects in Android apps. Most of these studies used apps downloaded from Google Play and extracted bytecode from the APK files. The extraction process includes a transformation process from DEX to Java bytecode. This transformation process generates a set

Table 1.1: Recent studies of Android apps analyzed aspects or purpose, number of apps, and number of Android categories covered.

Study	Purpose	#apps	#cat.	TPL	OBF
Shabtai *et al.* [70]	Apps categorization	2,285	2	NO	NO
Syer *et al.* [72]	Dependencies analysis	3	NR	YES	NI
Sanz *et al.* [67]	Apps categorization	820	7	NO	NO
Desnos [12]	Detection of similar apps	2	1	NO	NO
Mojica Ruiz *et al.* [57]	Reuse by inheritance and code cloning	4,323	5	NO	NO
Minelli and Lanza [53]	Visualization based analysis	20	NR	NI	NI
Mojica Ruiz *et al.* [56]	Reuse by inheritance and code cloning	$>$ 200K	30	NO	YES
Syer *et al.* [73]	Size, dependencies and defect fix time	15	NR	NO	NI
McDonnell *et al.* [41]	API instability and adoption	10	7	NI	NI
Linares-Vásquez *et al.*[36]	Apps success and API change/bug proneness	7,097	30	NI	NI

of .class files in a directory structure that follows the Java package guidelines. Therefore, the files belonging to third-party libraries and to the app's main package are organized using folders representing the packages hierarchy inside a single JAR file. In the following subsections we briefly describe the studies and summarize them in Table 1.1. In Table 1.1, we use NR to distinguish the cases where the number of domain categories is not reported. The last two columns list if the study considered the impact of third-party libraries (TPL) or the impact of obfuscated code (OBF): YES means the study considered the factor (NO is the opposite); NI stands for those cases where TPL and OBF factors do not impact the results.

1.1.1 Reuse in the Android Market

Mojica Ruiz *et al.* [57] were the first to report on the volume of reuse in Android apps. Two dimensions of reuse were analyzed: reuse by inheritance and class reuse (from other applications). About 4,000 Android apps were manually downloaded from Google Play to measure the percentage of classes that were totally reused (cloned) by other apps and the top base classes that were inherited from third-party libraries and platform APIs (Android and Java). Mojica Ruiz *et al.* [57] analyzed the reuse by class cloning in Android apps, by using class signatures as proposed by Davies *et al.* [8, 9]. The main conclusion of their study is that almost 50% of the classes in the apps inherit from a base class, and most of the reused classes are in the Android APIs. The same study was recently extended in [56] with more than 200K apps from GooglePlay. The results on the extended study showed that about 84% of the classes are reused across all the categories of apps. However, both studies included the code belonging to third-party libraries when measuring the percentage of class cloned in the apps; and only the latter [56] considered the impact of obfuscated classes. Desnos [12] also used method signatures to detect similar Android apps, where the signatures included string literals, API calls, exceptions, and control flow structures. However, the study does not report on the impact of obfuscated code or third-party libraries on their experiments.

Syer *et al*. [72] analyzed dependencies, source code, and churn metrics of three open source apps (i.e., Wordpress, Google Authenticator, and Facebook SDK) in Android and BlackBerry. Although they reported the findings in terms of apps dependency on pre-defined categories (e.g., language, user interface, platform, third-party), they analyzed different dimensions of reuse (i.e., inheritance, interface implementation, API calls) by counting the number of dependencies on each category and the proportion of platform and user interface dependencies out of the total number of dependencies. Their main conclusions were that Android apps require less source code but have larger files than in BlackBerry, and depend more on the Android APIs. During the analysis, the authors distinguished project-specific files from the source code of third-party libraries, and explicitly mentioned that *"apps often include, customize and maintain the source code of third party libraries"*[72].

Minelli and Lanza [53] proposed a visualization-based analysis for mobile apps using Samoa, which is an interactive tool that uses historical and structural information from the apps. Although the tool is not focused on a specific design aspect as reuse, the authors used the Average Hierarchy Height (AHH) and Average Number of Derived Classes (ANDC) metrics to study inheritance in Android apps. Moreover, they identified that some apps reuse libraries by copying the entire code instead of referencing JAR files. Some of the findings help to describe the programming model of Android apps (e.g., complexity of mobile apps is mostly attributed to the dependency on third-party libraries), however, only 20 open source apps were used in the study. Although the authors recognize the fact that the source code of third-party libraries is copied in some cases into the apps, they do not mention explicitly if the tool (Samoa) distinguishes between project-specific and third-party library files.

1.1.2 Other studies using Android apps

Syer *et al.* [73] analyzed 15 open source apps to investigate the differences of mobile apps with five desktop/server applications. The comparison was based on two dimensions: the size of the apps and the time to fix defects. The study suggests that mobile apps are similar to UNIX utilities in terms of size of the code and the development team. However, it is not clear if the analyzed apps included the source code of third-party libraries. Also, the findings suggest that mobile app developers are concerned with fixing bugs quickly: over a third of the bugs are fixed within one week and the rest are fixed within one month.

Categorization of Android applications has been explored using machine-learning techniques [67, 70]. Shabtai *et al.* [70] categorized APK files into two root categories of the Android market (``Games'' and ``Applications'') using attributes extracted from DEX files and XML data in the APK files. Sanz *et al.* [67] used string literals in classes, ratings, application sizes, and permissions to classify 820 applications into several existing categories. In both cases [67, 70], some of the extracted features could be obfuscated and could also belong to third-party libraries. Therefore it is possible that the results of the study were impacted by the effect of obfuscated code and third-party libraries.

McDonnell *et al* [41] analyzed the evolution of Android APIs (i.e., frequency of changes) and the reaction of client code to API evolution. For the latter purpose, they analyzed 10 open-source Android applications from 7 domains to investigate into: (i) degree of dependency on Android APIs; (ii) lag time between a client API reference and its most recent available version; (iii) adoption time of new APIs; (iv) the relation between API instability and adoption; and (v) the relationship between API updates and bugs in client code.

Also, Linares-Vásquez *et al.* [36] analyzed the impact of the Android APIs change- and fault-proneness on the success of 7,097 apps from Google Play. In both studies [36, 41], because they analyzed calls to the Android API, there was not an impact on the results by the effect of third-party libraries or obfuscated code.

1.2 Methodology

The *goal* of this study is to understand to what extent obfuscated code and third-party libraries could affect the studies on reuse by class cloning. The *context* consists of 24,379 free Android apps from the Google Play Market, and the *perspective* is that of researchers interested in defining guidelines for empirical studies based on Android apps. Table 1.2 reports characteristics of the apps that we analyzed. For each category considered in our study (e.g., photography, medical, games, etc), the table lists (i) the number of apps analyzed from the category (column #apps), (ii) the size range of the analyzed apps in terms of number of classes (column #classes), and size in terms of thousands of lines of code including third-party libraries (KLOC).

1.2.1 Research Questions

In the context of our study, we formulated the following research questions:

- **RQ**$_1$: *Do third-party libraries impact the measurement of class cloning?* This research question aims at investigating if the amount of class cloning in Android apps is mainly due to the dependability on the third-party libraries or the apps' classes. Specifically, we test the following null hypothesis:

  H_{0_1}*: There is no significant difference between the amount of cloned classes in Android apps when considering third-party libraries and when excluding those libraries from the analysis.*

Table 1.2: Characteristics of the apps (grouped by category) used in our study.

Category	#apps	Classes	KLOC
Arcade	826	5-566	625-20K
Books and reference	719	5-73	7K-639K
Brain	1021	5-572	5K-16K
Business	2047	5-551	64K-105K
Cards	495	8-633	30K-60K
Casual	840	6-566	60K-77K
Comics	57	10-392	251-20K
Communication	479	5-11	419-667K
Education	1572	5-119	9K-58K
Entertainment	2809	2-11	850-61K
Finance	586	5-1583	220-9K
Health and fitness	310	6-104	8K-26K
Libraries and demo	244	1-499	32K-338K
Lifestyle	1621	2-572	7K-16K
Media and video	644	5-572	8K-35K
Medical	102	5-105	6K-26K
Music and audio	1562	3-683	8K-14K
News and magazines	1015	5-280	26K-96K
Personalization	1055	2-126	12K-54K
Photography	595	6-155	111-31K
Productivity	639	5-111	11K-34K
Racing	456	15-280	26K-169K
Shopping	200	5-7	138-151K
Social	522	5-318	48K-122K
Sports	1158	5-280	7K-16K
Sports games	498	6-572	26K-52K
Tools	1421	4-65	7K-58K
Transportation	149	6-57	10K-202K
Travel and local	681	5-257	6K-16K
Weather	56	16-30	2K-22K
Total	**24,379**	**1-1583**	**111-667K**

- **RQ$_2$**: *Does obfuscated code impact the measurement of class cloning?* This research question aims at investigating if obfuscated apps should be considered when computing the amount of classes reused between Android apps. Specifically, we test the following null hypothesis:

  H_{0_2}: *There is no significant difference between the amount of cloned classes in Android apps when considering obfuscated apps and when excluding those apps from the analysis.*

The **dependent variable** for both research questions is represented by the amount of reuse by class cloning, which is estimated as the *Proportion of Class Signatures Reused (PCSR)* per category in our dataset (Section 1.2.3).

The **independent variable** for **RQ**$_1$ is the set of .class files of the apps under study including third-party libraries, and excluding those libraries. For **RQ**$_2$ the **independent variable** is the set of .class files of the apps under study including and excluding obfuscated apps.

1.2.2 Data Extraction Process

We downloaded free mobile apps from Google Play as APK files, then we converted the APK files into JAR files using the following procedure: (i) unzip APK files by using the *apktool*[1] tool, which reveals the compiled Android application code file (note that an APK is just a set of zipped DEX files); then (ii) translate DEX files from the Dalvik bytecode to Java bytecode files (i.e., .class) using the *dex2jar*[2] tool (see Figure 1.2).

Reuse by class cloning detection

For computing reuse via class cloning we relied on the Software Bertillonage technique [8, 9] to identify when a class is cloned across several apps, by comparing the classes' signatures. We built class signatures using the Apache Commons BCEL Java library[3] as in [8, 9]. Consequently, a class signature is a file with three parts: **class header**, **attributes signatures** sorted alphabetically, and **methods signatures** sorted alphabetically. The format of each part is as follows:

- The **class header** is defined by the following expression: <*modifiers*> <*class_name*> extends <*base_class*> implements <*interfaces_separated_by_comma*>. We avoided including the java.lang.Object class in the list of base classes.
- Each **attribute signature** is defined by the following expression: <*modifiers*> <*attribute_type*> <*attribute_ name*>.

[1]http://code.google.com/p/android-apktool/
[2]http://code.google.com/p/dex2jar/
[3]http://commons.apache.org/proper/commons-bcel/

```
public ZzActivity extends Activity implements View
private Button button1
public static int count
public int radioCheck
default static void <clinit>()
public void <init>()
public void onCheckedChanged(RadioGroup,int)
public void onClick(View)
public void onCreate(Bundle)
public boolean onKeyDown(int,KeyEvent)
```

Figure 1.1: Class signature example for the class zz.zzzz.ZzActivity in the Android zz.zzz App.

- Each **method signature** is defined by the following expression: <*modifiers*> <*return_type*> <*method_ name*> (<*argument_types*>).

The parts corresponding to the base class and interfaces are optional in the class header, and the names of types do not include the package in any of the parts. Figure 1.1 presents an example of a class signature.

In order to detect reuse by class cloning, we needed to find if any signature file's contents were exactly the same as the contents of another signature file. Even if we used certain optimizations on our comparisons to prevent redundant comparisons, it would be extremely time-consuming to repeatedly compare files directly. In order to overcome this obstacle, we opted to read in each signature file, and created an MD5 hash from the contents inside the signature. We created a large hash map which used the MD5 hash as the key, and contained a list of signature names for the value. For every signature file, we checked if the hash already existed as a key in the hash map. If it did, we appended the name of the signature file to the end of the list in the respective value. If not, we added the key/value pair to the hash map. Thus, once we finished adding every signature file's hash and name to the map, we were able to distinguish the cloned files from the originals.

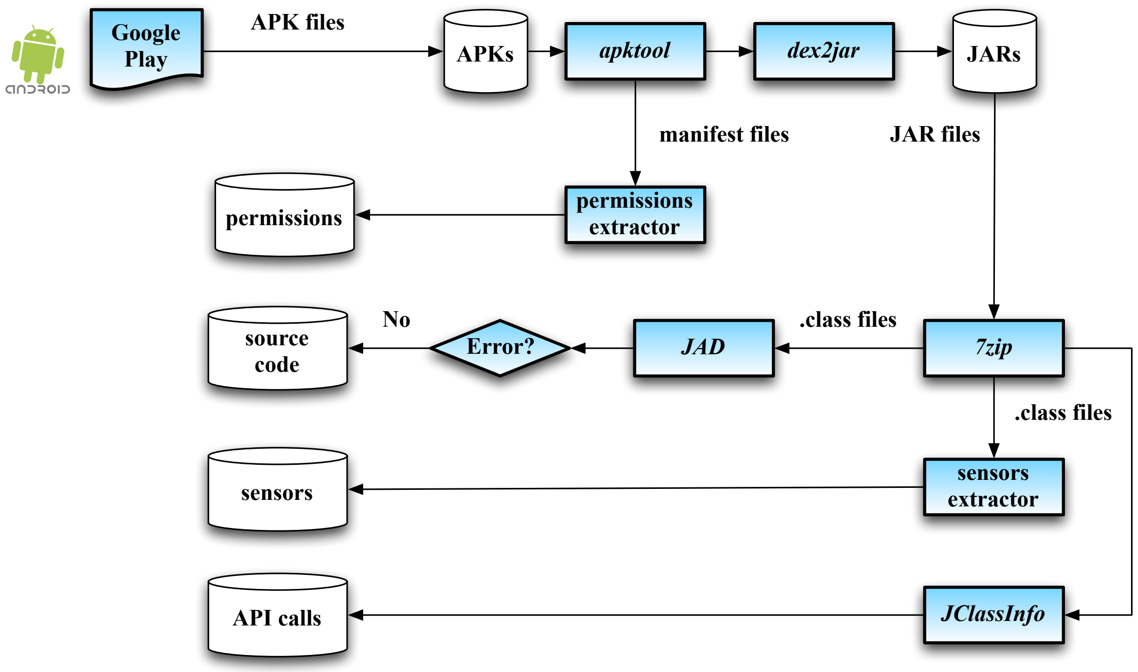


Figure 1.2: Source code and JAR files extraction process from APK files

Detecting obfuscated apps

One of the authors manually inspected the source code of 120 apps (i.e., two obfuscated and two non-obfuscated apps per category) to identify patterns in the identifiers of obfuscated classes. To decompile the apps we extracted .class files from the JAR files by using the *7zip*[4] tool and then we decompiled the .class files to Java source code using the JAD decompiler[5]. During decompilation, we discarded any apps that did not decompile correctly (see Figure 1.2). At the end, we were able to decompile 24,379 apps successfully.

After decompiling and manually inspecting the apps, we found that all the apps with obfuscated identifiers always have a class `a.java`, because of the renaming algorithm of the obfuscation tool used for Android apps transforms identifiers using a lexicographic order. Therefore, to detect apps with obfuscated identifiers we looked for apps with a class `a.java` in the main package. We decided to use this simple heuristic because we were interested only in the impact of identifier obfuscation in the class cloning estimation using

[4]`http://www.7-zip.org/`
[5]`http://www.varaneckas.com/jad/`

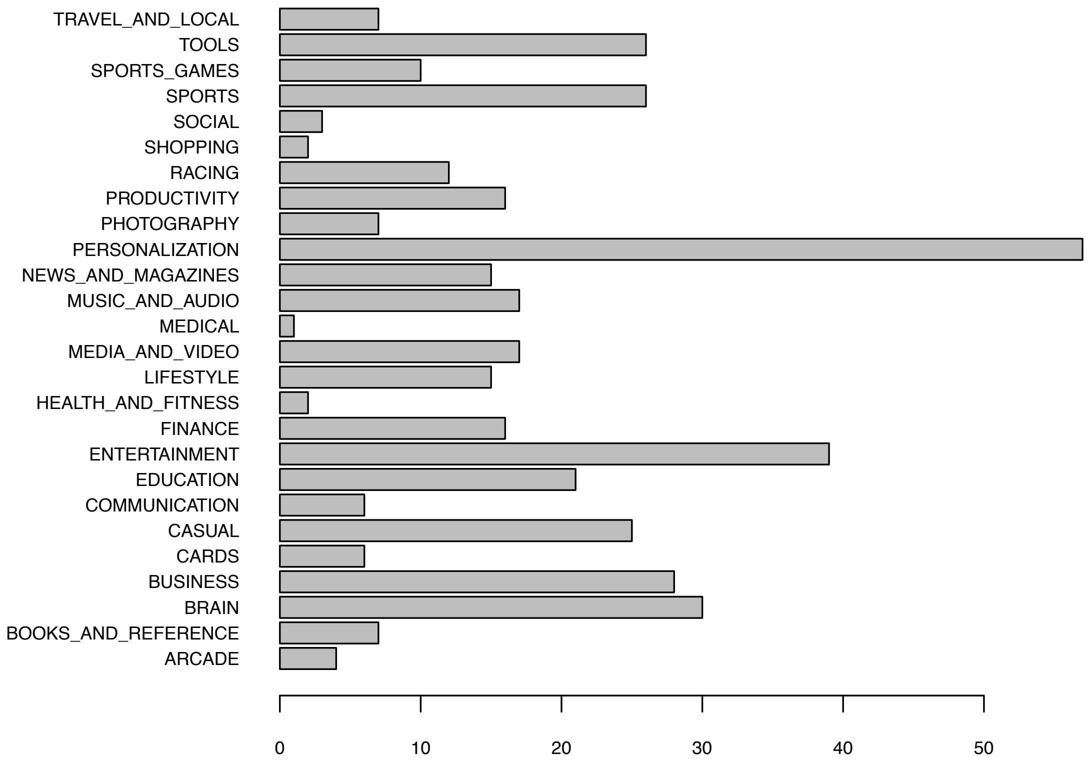


Figure 1.3: Distribution of obfuscated apps per category.

signatures. Using this method we found 415 apps with obfuscated code. The distribution of apps with obfuscated code per category is depicted in Figure 1.3.

To validate the accuracy of the method, another author of the paper manually verified the true positive rate (TPR) and false positive rate (FPR) of the heuristic for detecting obfuscated classes, by using a validation set of apps. The validation set was built using the following guidelines:

- The apps were sampled by one of the authors (not the same author performing the validation)
- The validation set includes two apps classified as obfuscated and two apps classified as non-obfuscated for each category (i.e., 120 apps).
- The apps in the validation set were different from the ones inspected manually for

identifying patterns in the identifiers of obfuscated classes

For the validation we followed these definitions[6]:

- **True positives (TP):** number of obfuscated apps classified correctly by the heuristic
- **True negatives (TN):** number of non-obfuscated apps classified correctly by the heuristic
- **False positives (FP):** number of non-obfuscated apps classified incorrectly by the heuristic (i.e., classified as obfuscated)
- **False negatives (FN):** number of obfuscated apps classified incorrectly by the heuristic (i.e., classified as non-obfuscated)
- **True positive rate (TPR), a.k.a., recall:** $TP/(TP + FN)$
- **True negative rate (TNR):** $TN/(FP/TN)$
- **Accuracy (ACC):** $(TP + TN)/(TP + TN + FP + FN)$

The results of the manual validation were 60 true positives and 60 true negative which accounts for a TPR equal to 1, a TNR equal to 1, and an ACC equal to 1. Therefore, our simple heuristic for detecting obfuscated apps is accurate and correct in a sample of 120 apps, which ensures a confidence interval of 8.93% with a confidence level of 95%.

1.2.3 Analysis Method

For measuring the amount of reuse by class cloning we used the *Proportion of Class Signatures Reused* ($PCSR$) proposed by Mojica Ruiz *et al.* [56, 57]. $PCSR$ calculates the proportion of class signatures that are clones (i.e., they appear in multiple apps belonging

[6]We were interested in the correctness of the heuristic for classifying apps in the positive set (i.e., obfuscated), and in the negative set (i.e., non-obfuscated), and in the general accuracy of the heuristic. Therefore, we used TPR, FPR, and ACC instead of precision

to a set of apps). Given a set of apps A, the number of Unique Class Signatures (UCS) in an app $a_i \in a$ ($a \subset A$), and C the set of all class signatures of the apps in a, the $PCSR$ of a subset of apps a is defined as follows:

$$PCSR(a, A) = 1 - \frac{\sum_{i=1}^{|a|} UCS(a_i, \{A - a_i\})}{|C|} \tag{1.1}$$

We defined a unique class signature in a_i as a signature that does not appear in the rest of apps in A ($\{A - a_i\}$ in equation 1.1). Thus, the higher the $PCSR$, the higher the reuse in a subset of apps (e.g., apps in the category *Arcade*) when compared to all the apps in A (e.g., all the apps in our dataset). Consequently, in order to compare the impact of third-party libraries on the measurement of reuse by class cloning (RQ$_1$) we computed the $PCSR$ per category (i.e., $PCSR$ of class signatures of apps belonging to a specific category that are cloned in all the 24,379 apps) including the class signatures of the third-party libraries ($PCSR_{+TPL}$); we also computed the $PCSR$ per category excluding class signatures of the third-party libraries ($PCSR_{-TPL}$). To compare the impact of obfuscated apps on the measurement of reuse by class cloning (RQ$_2$) we computed $PCSR$ per category excluding obfuscated apps ($PCSR_{-OBF}$), and excluding classes signatures of third-party libraries and obfuscated apps ($PCSR_{-(TPL,OBF)}$).

To validate that the results of our research questions are statistically significant in the 30 categories of Google play we used the Mann-Whitney test [13]. We compared $PCSR_{+TPL}$ to $PCSR_{-TPL}$ for H_{0_1}; and $PCSR_{-TPL}$ to $PCSR_{-OBF}$, and $PCSR_{+TPL}$ to $PCSR_{-(TPL,OBF)}$[7] for H_{0_2}. We also computed the Cliff's delta d effect size [25] to measure the magnitude of the difference in the three cases. We followed the guidelines in [25] to interpret the effect size values: negligible for $|d|$<0.147, small for $0.147 \leq |d|$<0.33, medium for $0.33 \leq |d|$<0.474 and large for $|d| \geq 0.474$. We are not assuming population normality and homogeneous variances, therefore we choose non-parametric methods (Mann-Whitney test and Cliff`s delta).

[7]Note that the apps used for computing $PCSR_{+TPL}$ and $PCSR_{-TPL}$ include obfuscated apps

1.2.4 Replication Package

The data set used in our study is publicly available at http://www.cs.wm.edu/semeru/data/MSR14-android-reuse/. In particular we provide: (i) the list (and URLs) of the studied 24,379 apps; (ii) the list of apps labeled manually as obfuscated and non-obfuscated; (iii) the dataset used for training the classifiers; and (iv) the results of the classification process and the manual validation.

1.3 Results

This section reports the results aimed at answering the two research questions formulated in Section 1.2.1. Table 1.3 summarizes the results for RQ$_1$ and RQ$_2$. In particular, the table lists the number of the proportion of class signatures reused ($PCSR$) in our dataset per category, when considering third-party libraries ($+TPL$), excluding third-party libraries ($-TPL$), excluding obfuscated apps ($-OBF$),and excluding third-party libraries and obfuscated apps (i.e., *-(TPL, OBF)*); Table 1.3 also lists the differences between the $PCSR$ values ($\Delta PCSR$). In addition, Figure 1.4 depicts the change ratio (i.e., reduction) of number of cloned signatures detected in the 30 categories, when comparing the initial dataset to the dataset without third-party libraries, and when comparing the dataset without third-party libraries to the dataset without libraries and without obfuscated apps.

1.3.1 Impact of third-party libraries

Excluding the libraries from the PCSR computation reduces notoriously the number of classes detected as clones. On average, 87.66% less signatures are detected as clones (see Figure 1.4 boxplot *+TPL to -TPL*), with a median of 90.70%, a minimum reduction of 67.08% (in the category *Health and fitness*), and a maximum reduction of 97.48% (in the category *Casual*). A similar behavior (i.e., reduction in all the categories) is reflected in the $PCSR$ computation (see Table 1.3). The average reduction of $PCSR$ when comparing

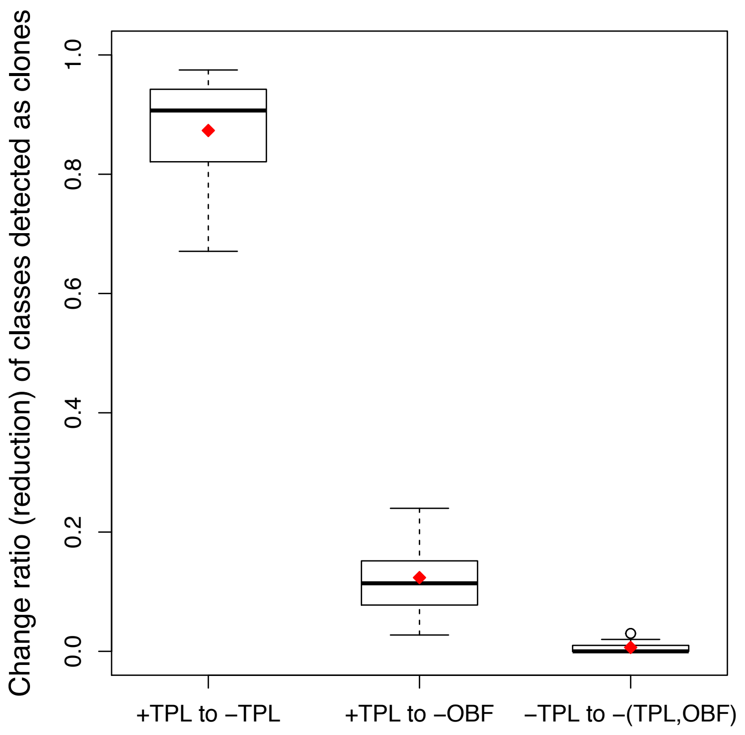


Figure 1.4: Boxplots for the change ratio of number of clones (signatures) when (1) comparing the dataset with third-party libraries and without third-party libraries (i.e., +TPL to -TPL); (2) comparing the dataset with third-party libraries, and the dataset without obfuscated apps (i.e., +TPL to -OBF); and (3) comparing the dataset without third-party libraries, and the dataset without third-party libraries and without obfuscated apps (i.e., -TPL to -(TPL, OBF)). Red diamonds represent the mean (average).

$PCSR_{+TPL}$ to $PCSR_{-TPL}$ is 37.30%, with a median of 37.94%, a minimum reduction of 7.45% (in the category *Business*), and a maximum reduction of 71.82% (in the category *Finance*).

That reduction in the number of class signatures detected as clones is large and significant. The Mann-Whitney test applied to the PCSR of signatures including third-party libraries ($PCSR_{+TPL}$) and the $PCSR$ of signatures excluding third-party libraries ($PCSR_{-TPL}$) reports a p-value= 9.123e-13, and the Cliff's delta was 0.9267 with a 95% confidence interval [0.8083, 0.9730]. Therefore, we can reject our null hypothesis H_{0_1}, that is, there is statistically significant difference between the two groups, and the magnitude of the difference is large (Cliff's delta > 0.474).

The significant reduction of the $PCSR$ and the number of clones when excluding the signatures of third-party libraries from the analysis shows that most of the clones are detected in the signatures of the libraries, and it suggests that most of the code in APK files belongs to the libraries. Figure 1.5 depicts the change ratio (i.e., reduction) of the number of class signatures when comparing the datasets including and excluding third-party libraries. On average, 82% of the signatures are reduced when excluding third-party libraries, with a median reduction of 81.23%, a minimum reduction of 63.82% (in the case of apps in the Category *Medical*), and a maximum reduction of 93.13% of the signatures (in the category *Arcade*).

We also analyzed which third-party library class signatures appeared most often, and attributed them to their respective third-party libraries. By doing so, we found that the most common third-party library class signature is `com.goo-gle.ads.AdActivity` of the `com.google.ads` package. This class is found in 8,008 apps from our dataset, and is by far the most common third-party library class. The second most common is `com.facebook.android.FacebookError`, from the `com.facebok.android` package. This class signature was found in 6,652 apps. Finally, the third most common is `org.mcsoxford .rss.Dates` (and 22 other classes from this same package), which all appeared 4,880 times each.

A significant number of apps in our dataset utilize the Google Ads third-party library, potentially as a source of revenue due to all the apps in our dataset being free. Also free apps have the option for Facebook integration. Finally, the commonality of `org.mcsoxford.rss` demonstrates that many apps try to integrate with RSS feeds, and this third-party library is described as a "lightweight Android library to read parts of RSS 2.0 feeds." [8]

[8]https://github.com/ahorn/android-rss

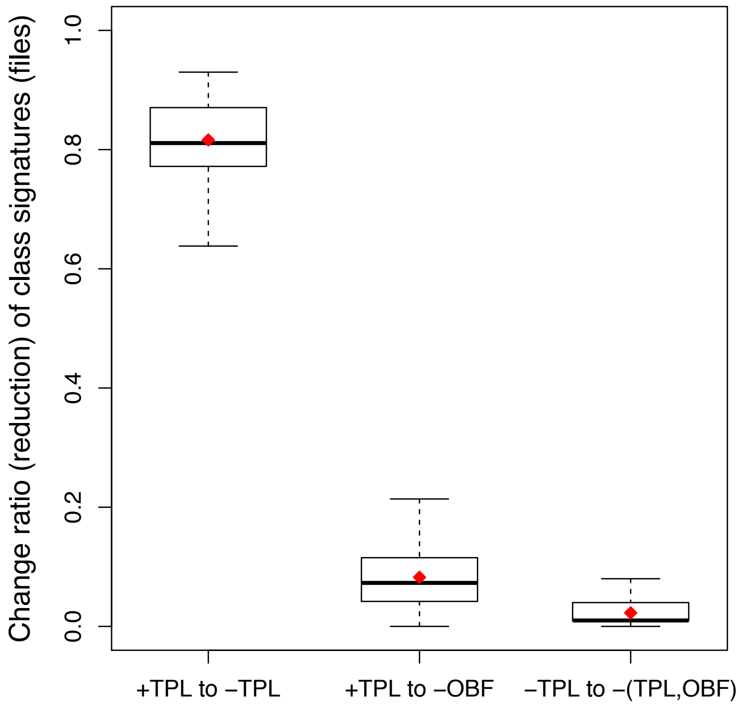


Figure 1.5: Boxplots for the change ratio of number of signatures when (1) comparing the dataset with third-party libraries and without third-party libraries (i.e., +TPL to -TPL); (2) comparing the dataset with third-party libraries, and the dataset without obfuscated apps (i.e., +TPL to -OBF); and (3) comparing the dataset without third-party libraries, and the dataset without third-party libraries and without obfuscated apps (i.e., -TPL to -(TPL, OBF)). Red diamonds represent the mean (average).

Summarizing, the results of our **RQ**$_1$ shows that *considering third-party libraries when computing class cloning in Android apps impacts the results, in the sense that because of the wide usage of third-party libraries, a significant number of clones are detected between the apps. Therefore, an actionable guideline when analyzing APK files is: consider carefully if third-party libraries should be included or not in the specific analysis; in particular, when analyzing class cloning between Android apps, researchers should expect the amount of clone detection to be inflated if third-party libraries are included in the dataset, while the exclusion of third-party libraries will lower this amount of clone detection.*

Table 1.3: Summary of results for RQ$_1$ and RQ$_2$. The PCSR and the difference between PCSR are listed by category.

Category	$PCSR$				$\Delta PCSR$ in percentage		
	$(1) + TPL$	$(2) - TPL$	$(3) - OBF$	$(4) - (TPL, OBF)$	$\frac{(1)-(2)}{(1)}$	$\frac{(1)-(3)}{(1)}$	$\frac{(2)-(4)}{(2)}$
Arcade	0.879688807	0.351430128	0.829749587	0.353	60.05%	5.68%	-0.49%
Books and reference	0.902051778	0.764008747	0.865465015	0.766	15.30%	4.06%	-0.32%
Brain	0.887832825	0.430272179	0.856644088	0.443	51.54%	3.51%	-2.98%
Business	0.87552092	0.81029478	0.83328423	0.819	7.45%	4.82%	-1.03%
Cards	0.839701923	0.383950449	0.803221733	0.394	54.28%	4.34%	-2.52%
Casual	0.879285526	0.316837771	0.848328398	0.334	63.97%	3.52%	-5.29%
Comics	0.920239358	0.569789675	0.895015907	0.570	38.08%	2.74%	0.00%
Communication	0.721917416	0.267079672	0.698035799	0.278	63.00%	3.31%	-4.02%
Education	0.930289647	0.752647301	0.89829617	0.758	19.10%	3.44%	-0.70%
Entertainment	0.926801039	0.778399296	0.890272919	0.782	16.01%	3.94%	-0.45%
Finance	0.73930074	0.2083375	0.671912979	0.214	71.82%	9.12%	-2.72%
Health and fitness	0.942137572	0.858483567	0.910759035	0.858	8.88%	3.33%	0.06%
Libraries and demo	0.945508264	0.58816772	0.906561089	0.588	37.79%	4.12%	0.00%
Lifestyle	0.895974257	0.670682596	0.854727031	0.673	25.14%	4.60%	-0.39%
Media and video	0.82493961	0.350162866	0.778602469	0.356	57.55%	5.62%	-1.57%
Medical	0.892509122	0.79831534	0.868709734	0.797	10.55%	2.67%	0.14%
Music and audio	0.876889856	0.772113587	0.872644424	0.782	11.95%	0.48%	-1.30%
News and magazines	0.898420806	0.570090694	0.863682988	0.579	36.55%	3.87%	-1.53%
Personalization	0.916485781	0.579639994	0.882627703	0.597	36.75%	3.69%	-2.97%
Photography	0.841369352	0.488375841	0.802766729	0.493	41.95%	4.59%	-0.93%
Productivity	0.760950939	0.372573998	0.697382465	0.386	51.04%	8.35%	-3.65%
Racing	0.92630846	0.544961203	0.914853858	0.570	41.17%	1.24%	-4.66%
Shopping	0.778630803	0.241201949	0.722952071	0.248	69.02%	7.15%	-2.80%
Social	0.891405177	0.769676122	0.874447338	0.771	13.66%	1.90%	-0.20%
Sports	0.913609539	0.70980359	0.8654784	0.721	22.31%	5.27%	-1.52%
Sports games	0.928972353	0.515475313	0.900676647	0.520	44.51%	3.05%	-0.81%
Tools	0.798852806	0.423019698	0.719613626	0.439	47.05%	9.92%	-3.86%
Transportation	0.818652745	0.396924049	0.755777412	0.397	51.51%	7.68%	0.00%
Travel and local	0.913801067	0.727011219	0.859122395	0.727	20.44%	5.98%	0.06%
Weather	0.949381457	0.660390516	0.90783172	0.660	30.44%	4.38%	0.00%

1.3.2 Impact of obfuscated apps

Excluding obfuscated apps also reduced the number of signatures detected as clones, and consequently $PCSR$. The Mann-Whitney test applied to the PCSR of signatures including third-party libraries ($PCSR_{+TPL}$) and the $PCSR$ of signatures excluding obfuscated apps ($PCSR_{-OBF}$) reports a p-value= 0.009604, and the Cliff's delta was 0.3866667 with a 95% confidence interval [0.08998, 0.62031998]. Therefore, we can reject our null hypothesis H_{0_2}, i.e. there is a statistically significant difference between the two groups, and the magnitude of the difference is medium ($0.33 \leq |d| < 0.474$).

On average (see Table 1.3) there is a reduction of 4.55% in the $PCSR$, with a median of 4.09%, a minimum reduction of 0.48% (*Music and Audio*), and a maximum reduction of 9.92% (*Tools*). This medium reduction (in terms of effect size) is explained due to the number of signatures belonging to obfuscated apps (we found 415 obfuscated apps out of 24,415). When excluding obfuscated apps (see Figure 1.5) 8.25% of the signatures were reduced on average (median = 7.31%, min. = 0%, max=21.38%), which represented an average reduction in the number of signatures detected as clones (see Figure 1.4) of 12.40% (median = 11.43%, min. = 2.74%, max=23.98%).

However, when comparing the impact of obfuscated code in the $PCSR$ excluding the signatures of third-party libraries (i.e., -TPL to -(TPL, OBF)) the Mann-Whitney reports a p-value=0.8187, and we obtained a Cliff's delta = -0.0356. In this case there is no significant difference (p-value $>$ 0.05) and the magnitude of the difference is negligible ($|d| < 0.147$). When removing the obfuscated apps from the set of signatures that does not include third-party libraries there is an average reduction in the number of signatures detected as clones of 0.63% (median = 0%, min. = 0%, max=3%), and an average reduction in the number of signatures of 2.23% (median = 1%, min. = 0%, max=8%). However, in most of the categories (23 out of 30) removing the obfuscated apps increases the $PCSR$ (see Table 1.3). For example, there is a change in the $PCSR$ of the category *Casual* from 0.3168 ($PCSR_{-TPL}$) to 0.334 ($PCSR_{-(TPL,OBF)}$), which accounts for an increment of 5.29%.

An explanation for those cases is the impact of the reduction of the signatures in the $PCSR$ computation. Equation 1.1 is equivalent to the ratio between the number of signatures detected as clones and the total number of signatures. In the case of apps in the category *Casual* for $PCSR_{-TPL}$ there were 11,851 signatures detected as clones out of 37,404 signatures ($PCSR_{-TPL} = 11,851/37,404 = 0.3168$), and for $PCSR_{-(TPL,OBF)}$ there were 11,539 signatures detected as clones out of 34,589 signatures ($PCSR_{-(TPL,OBF)} = 11,539/34,589 = 0.334$). That increment of 5.29% in the $PCSR$ is explained in the fact that proportionally the reduction of the signatures is bigger compared to the reduction of clones, which means that most of the clones were detected between the non-obfuscated apps. However, there were some signatures detected as clones between the obfuscated apps.

Regarding detecting cloned classes in the dataset of apps tagged as obfuscated, we inspected manually the signatures and we found that there are some false positives.

That is, there are classes that are marked as clones of other classes based on their class signatures, but further analysis of the content of the class demonstrated that this is not always true. We were able to find multiple examples of this occurring fairly easily, and we believe that there could be many more false clone detections in our obfuscated dataset as a result of this observation. The first example comes from the apps with package names *bagins.football* and *com.antivirus*. In both apps we found two obfuscated classes that were detected as cloned signatures: `/bagins/football/c/c.java` and `/com/antivirus/core/b/c.java`. Both of these files have the same signatures and thus method names, but these methods do different things. For instance, in *bagins.football* the values() method creates a new array and performs a `System.arraycopy` into it, whereas in *com.antivirus* the method only has a statement returning a casted variable with `.clone()`.

Another example we found is between the apps *com.agilesoft resource* and *com.ableon.team.barcelona*. Both apps have a class called `h.java` inside their main package, and both classes have a `void run()` method. However, the run function in `h.java` of *com.agilesoftresource* is simply a one-line statement:

```
"AppManagerMain.a(AppManagerMain.e(g.a(a))). refreshPackList();"
```

whereas the `run` method of *com.ableon.team.barcelona* is 15 lines long and makes calls to the `javax.microedition.khronos.egl` API and performs an obfuscated conditional:

```
"if(a.isVisible() && g.c(a).eglGetError() = 12302)"
```

Some cloned classes appear in more than two apps. One such example is the class `as.java`, which appears in three apps: *balofo.game.movie*, *com.application.fotodanz*, and *com. advancedprocessmanager*. For each app, this class has the `onClick()` method, but the code it executes is unique in each case. For *balofo.game.movie*, the method simply performs a `dialoginterface.cancel()`; for *com.application.fotodanz*, the method executes no code; and for *com.advancedprocess manager*, we get an "obfuscated" one-line of code:

```
"ak.a(aq.a(a)).a()"
```

The examples described before show how using class signatures to detect clones between obfuscated classes is not accurate because it is prone to false positives. However, there are also cases of true positives. We have noted that if apps share a main package or developer "keyword" in the app's package name then it is likely that the files are indeed clones. For instance, *com.appmakr.app247821* and *com.appmakr.app153560* both have a class called `c.java` that were located within different directories inside each main package respectively, but were still detected as clones. Due to each app's package name sharing the term *appmakr*, we assume it's likely for these two files to be legitimate clones; upon further inspection, each file is 43 lines long and both files are the exact same, character for character, except for one line which references the main package name (*app247821* or *app153560* respectively). Therefore, these files are correctly detected as legitimate clones.

Finally, we should note that because we're only trying to find cloned classes that reside within the main package of the app, we've extracted the package name from the `AndroidManifest.xml` file that resided with every application we downloaded. Thus, sometimes a cloned class may appear to lie in a package different from the source directory, but is in fact within the proper main package. For instance, another cloned class was `q.java` which appears in apps *com.atomimbh.app*, *com.BeltzandRuth*, and *com.bangladeshfreegoimbh.app*. We noted that both the first and last apps in this list seem to follow the trend of having a shared keyword (`*imbh.app`), but one of the apps doesn't follow this pattern. However, for this app the actual location of this cloned class is found in *com.BeltzandRuth/src/ com/bemyvalentineimbh/app/*, which does share the similar keyword as the other two apps. Upon analyzing the Android manifest for this app, the main package is indeed *com.bemyvalentineimbh.app*. Thus, upon further inspection of the `q.java` class, we note that all three apps have a similar implementation of both methods inside the class, where both *com.BeltzandRuth* and *com.bangladeshfreegoimbh.app* contained exactly the same implementations, and *com.atomimbh. app* contained the exact implementation of one method and a functionally similar implementation of another method (only a few lines had their order changed).

Summarizing, the results of our **RQ**$_2$ shows that *considering obfuscated apps when computing class cloning in Android apps impacts the results, in the sense that signatures in obfuscated classes introduce false positives in the cloned signatures detection. Although the impact of obfuscated code is not as significant as the impact of considering third-party libraries in the cloned signatures detection, researchers should be careful when considering obfuscated code in their experiments using APK files. Therefore, an actionable guideline when analyzing APK files is: consider carefully if obfuscated apps (or obfuscated code) should be included or not in the specific analysis; in particular, when analyzing class cloning between Android apps, researchers should expect the amount of clone detection to be inflated if obfuscated apps are included in the dataset, while the exclusion of obfuscated apps will lower this amount of clone detection.*

1.4 Threats to Validity

Threats to construct validity concern the relationship between theory and observation, and it is essentially due to the measurements/estimates on which our study is based. We assumed that class signatures are representative of the actual source code files as in previous studies [8, 9, 56, 57]. However, we cannot state that the code inside the source files is exactly the same based solely on matching signatures. Instead, the methods may have been named similarly or may have had the same parameters. Therefore, it is likely that there is much more source code reuse occurring that we have been unable to detect in the case of class cloning. As this is an initial study of reuse, for future work we plan to obtain more exact results, by considering also the source code.

Threats to *conclusion validity* concern the relationship between treatment and outcome. Our conclusions are supported by appropriate, non-parametric statistics (Mann-Whitney test). In addition, the practical relevance of the observed differences is highlighted by effect size measures (Cliff's delta).

Threats to internal validity concern factors that can affect our results. Our heuristic for identifying obfuscated apps could fail if the renaming strategy did not follow a lexicographic order (i.e., the first letted used to obfuscate identifiers is *a*) or the obfuscation is different to renaming transformation. However, we manually inspected a sample of apps classified by the heuristic and we obtained a true positive and true negative rates equals to 1, which represents an accuracy of 100%.

Threats to *external validity* concern the generalization of our findings. Our analysis is limited to Android free apps that use a revenue model based on advertisements. Dependency of commercial apps on third-party libraries could be different, for example, libraries for advertisements might not be widely used in commercial apps. Also, it is possible that the commercial apps have more obfuscated code. Therefore, our findings may not necessarily hold for commercial apps. Regarding the size of our dataset (24,379 apps), the set of analyzed apps is a small percentage of the existing apps in Google Play (more than 1 million of apps reported by the AppBrain website[9]). However, our sample covers all the domain categories in Google Play with a significant number of apps compared to other studies using Android apps (see Table 1.1). In future studies, we are also planning on using diversity measures to guide the selection of apps to maximize generalizability of the case studies [59]. Finally, our conclusions may not be valid for apps developed for other mobile platforms (e.g., iOS).

[9]`http://www.appbrain.com/stats/number-of-android-apps`

1.5 Discussion

Although APK files have been used in several studies for analyzing Android apps and their development processes, the building process used to generate those files introduces some threats to the validity of the results in the studies. In particular, we analyzed 24,379 APK files downloaded from Google Play to measure the impact of third-party libraries and obfuscated code on class cloning measurement. We found that excluding third-party libraries reduces on average 87.66% of the signatures detected as clones, and the difference is large and statistically significant when comparing the proportion of class signatures reused (PCSR) in our dataset including and excluding the libraries. Concerning the impact of obfuscated files, it is significantly different but the difference is medium on the computation of the PCSR. We found a few of the obfuscated apps and evidence of false positives detected as clones by the signature-based method (Software Bertilonage). Future studies with significantly higher number of obfuscated apps should analyze the impact of those apps on the results.

Our findings show that empirical studies using APK files should take into account possible impacts of third-party libraries and obfuscated code. Therefore, we suggested two actionable guidelines when analyzing/mining APK files:

1. *Consider carefully if third-party libraries should be included or not in the specific analysis; in particular, when analyzing class cloning between Android apps, researchers should justify the decision of including/excluding third-party libraries libraries in the class cloning measurements. Researchers should expect the amount of clone detection to be inflated if third-party libraries are included in the dataset, while the exclusion of third-party libraries will lower this amount of clone detection.*

2. *Consider carefully if obfuscated apps (or obfuscated code) should be included or not in the specific analysis; in particular, when analyzing class cloning between Android apps, researchers should justify the decision of including/excluding obfuscated code*

in the class cloning measurements. Researchers should expect the amount of clone detection to be inflated if obfuscated apps are included in the dataset, while the exclusion of obfuscated apps will lower this amount of clone detection.

These actionable guidelines are also pertinent to studies/approaches on software categorization [31, 38, 43], in which the lexical information in bytecode or source code is used to categorize the apps; given the widespread use of third-party libraries, such as Google Ads or Facebook for Android using the identifiers extracted from those libraries can reduce the variance and consequently impact the categorization process. In addition, studies aimed at identifying similar apps [45], which use non-textual based detection, should also consider the impact of third-party libraries and obfuscation practices.

Chapter 2

Detecting Similar Android Applications

2.1 Introduction

NOTE: CLAN was previously published in ICSE 2012 [45] and as a part of Dr. Collin McMillan's dissertation [44]. CLAN is presented for completeness and comprehensive purposes, as CLANdroid extends this approach and evaluation to Android and is the contribution of this thesis. Sections 2.1 to 2.3 contain some content that was previously published as part of CLAN, but also contains new content related to CLANdroid.

Developers and end-users take advantage of code search engines, for browsing and searching software systems that are relevant to their needs. In the case of developers, the motivation could be opportunistic reuse, market analysis (i.e., finding similar applications to a system under development), prototyping, or simply looking for a tool that supports development processes. In the case of end-users, the motivation could be as simple as looking for a tool that supports a daily activity, or from an economic viewpoint, users could look for substitutes or complementary goods (i.e., similar tools, and tools that need to be used complementary). These scenarios apply to systems with different sizes. For example, users of mobile applications take advantage of mobile applications markets, (e.g., Google Play, iOS Market, Windows Phone Market, FDroid) for browsing and searching apps.

Knowing similarity between applications plays an important role in assessing reusability of these applications, improving understanding of source code, rapid prototyping, and discovering code theft and plagiarism [33, 39, 51, 65, 68, 42, 61, 50]. Enabling programmers to compare automatically how different applications implement the same requirements greatly contributes to knowledge acquisition about these requirements and subsequently to decisions that these developers make about code reuse. Retrieving a list of similar applications provides a faster way for programmers to concentrate on relevant aspects of functionality, thus saving time and resources for programmers. Programmers can spend this time understanding specific aspects of functionality in similar applications, and see the complete context in which the functionality is used.

Furthermore, having a list of similar applications has become especially relevant due to the popularity of mobile devices and the distribution of mobile applications. In order to keep users interested in applications, mobile application marketplaces, such as Google Play and the Apple App Store, commonly display similar applications based on which application the user is viewing. Thus, when a user is searching for an application to accomplish some functionality, if this user finds a fitting application which achieves the needed purpose, it is likely that this user would want to view a similar application in order to choose the application that suits the needed functionality best.

In general, retrieving relevant applications and code snippets starts with a search query submitted to a search engine, which displays the relevant code units (i.e., system, package, method, class, etc). However, detecting similar applications is a notoriously difficult problem, since it means automatically detecting that high-level requirements for these applications match semantically [29, pages 74,80][40]. This situation is aggravated by the fact that many application repositories are polluted with poorly functioning projects [27]; a match between words in requirement documents with words in the descriptions or in the source code of applications does not guarantee that these applications are relevant to the requirements. Applications may be highly-similar to one another at a low-level of the implementations of some functions even if they do not perform the same high-level functionality [15]. One example of an app which has very few legitimately similar apps on its app page is *Star Solitaire*[1]. This app is a card game with options to play different forms of solitaire, and it is likely misclassified to the wrong category - it is currently in `Strategy` when it should belong in `Cards`. Thus, because Google Play only lists similar apps from the same category, we see "similar" apps such as *Star Wars Force Collection*[2], *Star Colonies*[3], and *Star Girl: Beauty Queen*[4].

A fundamental problem of detecting closely related applications is in the mismatch between the high-level intent reflected in the descriptions of these applications and low-level implementation details. This problem is known as the *concept assignment problem* [3]. For any two applications it is too imprecise to establish their similarity by simply matching words in the descriptions of these applications, comments in their source code, and the names of program variables and types. Since programmers typically invest a significant intellectual effort (i.e., they need to overcome a high cognitive distance [34]) to understand whether retrieved applications are similar, existing code search engines do not alleviate the task of detecting similar applications because they return only a large number of different code snippets.

To overcome the *concept assignment problem* in the particular case of finding similar apps, we created an approach for detecting closely related Android applicatons. This approach is based on CLAN, a previously published approach for detecting similar Java applications. As our approach is a modification of CLAN, we name our approach CLANdroid. *CLANdroid* uses complete mobile Android applications as input, and outputs related Android applications. Although McMillan *et al.* [45] used *Application Programming Interface (API)* calls and source code identifiers as two differing methods of detecting sim-

[1]https://play.google.com/store/apps/details?id=com.kiwifruitmobile.solitaire
[2]https://play.google.com/store/apps/details?id=jp.konami.swfc
[3]https://play.google.com/store/apps/details?id=com.blueplop.starcolonies
[4]https://play.google.com/store/apps/details?id=com.animoca.google.starGirlBeautyQueen

ilar applications, we extend these methods to attributes unique to Android applications: Android intents, user permissions, and sensors usage.

In this thesis, we extend the *CLAN* approach to the Android ecosystem (*CLANdroid*). Similarly to the intuition behind the concept of using API calls as semantic anchors to compute similarities between software applications, we apply this same idea with unique features of Android applications: explicit and implicit intents used in the apps, user permissions declared in the manifest files, and sensors used by the apps declared in the source code. Therefore, for *CLANdroid*, we expand upon the new abstraction introduced by CLAN by not only using APIs, but also features unique to Android applications such as intents, user permissions, and sensors. In addition, following the guidelines presented by Linares-Vásquez *et al.* [37], we analyzed the impact of third-party libraries and obfuscated apps when detecting similar apps using APK (Android PacKage) files. Results in this thesis demonstrate that, conversely to Java systems, identifiers extracted from Android apps outperforms Android-specific semantic anchors when detecting similar apps. Our findings confirms the results in previous studies that suggest that Android apps are highly dependent on the Android SDK [56, 53], in the sense that API calls should be combined with other semantic anchors or attributes (e.g., identifiers) for detecting similar apps; API calls in Android apps are not enough to identify variability across different apps. Also, we found that third-party libraries and obfuscated code impacts significantly the detection of similar Android apps.

This thesis makes the following contributions:

- An approach for detecting similar mobile Android applications, which is useful for developers and users when browsing and searching applications. We implemented this approach in CLANdroid and applied to a set of 14,450 free Android applications that were downloaded from Google Play.

- To evaluate CLANdroid, we used a goldset of similar apps which we obtained from the Google Play market. We then compared each relevancy ranking method (APIs, identifiers, intents, etc.) to see which ranking method detects similar applications the best. We found that when considering the whole dataset, the identifiers ranking method is most effective.

- An online version of *CLANdroid* that can be used to list similar Android apps, using different datasets (i.e., including third-party libraries and obfuscated apps, excluding third-party libraries, and excluding obfuscated apps), and different approaches (i.e., identifiers, API calls, intents, sensors, user permissions).

2.2 Hypothesis And the Problem

In this section we use a conceptual framework for relevance to define the concept of similarity between applications, formulate a hypothesis, and describe problems that we should solve to test this hypothesis.

2.2.1 A Motivating Scenario[5]

A motivating scenario for detecting similar application is based on a typical project lifecycle in Accenture, a global software consulting company with over 250,000 employees as of February, 2012. At any given time, company consultants are engaged in over 3,000 software projects. Since its first project in 1953, Accenture's consultants delivered tens of thousand of projects, and many of these projects are similar in requirements and their implementations. Knowing the similarity of these applications is important for preserving knowledge, experience, winning bids on future projects, and successfully building new applications.

A typical lifecycle of a large-scale project involves many stages that start with writing a proposal in response to a bid from a company that needs an application. A major part of writing a proposal and developing a prototype is to elicit requirements from different

[5]Some of the material in this section was previously published in ICSE 2012 [45] and as a part of Dr. Collin McMillan's dissertation [44].

stakeholders. There are quite a few competing companies for each bid: IBM Corp, HP Corp, Tata Consultancy Services to name a few. A winning bid proposal has many components: well-elicited requirements, preliminary models and design documents, proof of experience of building and delivering similar applications in the past. Clearly, a company that submits a bid proposal that contains these components as closely matching a desired application as possible, will win the bid.

It is important to reuse these components from successfully delivered applications in the past - doing so will save time and resources and increase chances of winning the bid. It is shown that over a dozen different artifacts can be successfully reused from software applications [30, pages 3--5]. The process of finding similar applications starts with code search engines that return code fragments and documents in response to queries that contain key words from elicited requirements. However, returned code fragments are of little help when many other non-code artifacts are required (e.g., different (non)functional requirements documents, UML models, design documents).

Matching words in queries against words in documents and source code is a good starting point, however, it does not help stakeholders to establish how applications are similar at a bigger scale. In terms of the work presented in this thesis, we refer to an *application* as a collection of all source code modules, libraries, sensors, permissions, and programs that, when compiled, result in the final deliverable that customers install and use to accomplish certain functions. Applications are usually accompanied by non-code artifacts, which are important for the bidding process. Establishing their similarity at large from different similar components of the source code is a goal of this thesis.

The concept of similarity between applications is integrated in the software lifecycle process as follows. After obtaining the initial set of requirements, the user enters keywords that represent these requirements into a search engine that returns relevant applications that contain these keywords. In practice, it is unlikely that the user finds an application that perfectly matches all the requirements - if it happens, then the rapid prototyping process is finished. Otherwise, the user takes the returned applications and studies them to determine how relevant they are to the requirements.

After examining some returned application, the user determines what artifacts are relevant to requirements, and which ones are missing. At this point the user wants to find similar applications that contain the missing artifacts while retaining similarity to the application that the user has found. That is, using the previously found application, the initial query is further expanded to include artifacts from this application that matched some of requirements as the user determined, and similar applications would contain artifacts that are similar to the ones in the found application.

2.2.2 Similarity Between Applications[6]

We define the meaning of similarity between applications by using Mizzaro's well-established conceptual framework for relevance [54, 55]. In Mizzaro's framework, similar documents are relevant to one another if they share some common concepts. Once these concepts are known, a corpus of documents can be clustered by how documents are relevant to these concepts. Subsequently all documents in each cluster will be more relevant to one another when compared to documents that belong to different clusters. This is the essence of the cluster hypothesis that specifies that documents that cluster together tend to be relevant to the same concept [75].

Two applications are similar to each other if they implement some features that are described by the same abstraction. For example, if some applications use cryptographic services to protect information then these applications are similar to a certain degree, even though they may have other different functionalities for different domains. Another example is text editors that are implemented by different programmers, but share many features: copy and paste, undo and redo, saving data in files using standard formats. A straightforward approach for measuring similarity between applications is to match the names of their program variables and types. The precision of this approach depends

[6]Some of the material in this section was previously published in ICSE 2012 [45] and as a part of Dr. Collin McMillan's dissertation [44].

highly on programmers choosing meaningful names that reflect correctly the concepts or abstractions that they implement, but this compliance is generally difficult to enforce [1].

2.2.3 Our Hypothesis[7]

In Mizzaro's framework, a key characteristic of relevance is how information is represented in documents. We concentrate on *semantic anchors*, which are elements of documents that precisely define the documents' semantic characteristics. Semantic anchors may take many forms. For example, they can be expressed as links to web sites that have high integrity and well-known semantics (e.g., cnn.com or whitehouse.gov) or they can refer to elements of semantic ontologies that are precisely defined and agreed upon by different stakeholders.

This is the essence of *paradigmatic associations* where documents are considered similar if they contain terms with high semantic similarities [64]. Our hypothesis is that by using semantic anchors it is possible to compute similarities between documents with a higher degree of accuracy when compared to documents that have no commonly defined semantic anchors in them.

Without semantic anchors, documents are considered as bags of words with no semantics, then the relevance of these documents to user queries and to one another can be determined by matches between these words. This is the essence of *syntagmatic associations* where documents are considered similar when terms (i.e., words) in these documents occur together [64]. For example, the similarity engine MUDABlue uses syntagmatic associations for computing similarities among applications [32]. Although the original *CLAN* approach found this approach to be relatively imprecise, we find that this approach surprisingly works well in regards to Android applications as seen in Section 2.4.5.

[7]Some of the material in this section was previously published in ICSE 2012 [45] and as a part of Dr. Collin McMillan's dissertation [44].

2.2.4 Semantic Anchors in Software[8]

Since programs contain code elements (e.g., API calls, user permissions) with precisely defined semantics, these code elements can serve as semantic anchors to compute the degree of similarity between applications by matching the semantics of these applications that is expressed with these elements. Programmers routinely use API calls from third-party packages (e.g., the *Java Development Kit (JDK)*) to implement various requirements [4, 11, 22, 24, 46, 23, 71]. API calls from well-known and widely used libraries have precisely defined semantics unlike names of program variables and types and words that programmers use in comments. In this thesis, we use API calls as semantic anchors to compute similarities among mobile applications. However, we extend upon CLAN by not only using API calls as semantic anchors, but also using Android intents, user permissions, and sensors. Android intents are a unique part of the API provided by the Android OS for developers, which are used for declaring and reusing operations. According to the official reference guide for intents, an intent is "basically a passive data structure holding an abstract description of an action to be performed" [17]. A permission for an Android app is the "mechanism that enforces restrictions on the specific operations that a particular process can perform" [20]. The sensors for a phone can utilize the "built-in sensors that measure motion, orientation, and various environmental conditions" [19].

2.2.5 Challenges[9]

Our hypothesis is based on our idea that it is better to compute similarity between programs by utilizing semantic anchors that come from the JDK and Android SDK, and that programmers use to implement various requirements. This idea has advantages over using *Vector Space Model (VSM)* where documents are represented as vectors of words and a similarity measure is computed as the cosine between these vectors [66]. One main

[8]Some of the material in this section was previously pufblished in ICSE 2012 [45] and as a part of Dr. Collin McMillan's dissertation [44].

[9]Some of the material in this section was previously published in ICSE 2012 [45] and as a part of Dr. Collin McMillan's dissertation [44].

problem with VSM is that different programmers can use the same words to describe different requirements (i.e., the synonymy problem) and they can use different words to describe the same requirements (i.e., the polysemy problem). This problem is a variation of the vocabulary problem, which states that "no single word can be chosen to describe a programming concept in the best way" [14]. This problem is general to *Information Retrieval (IR)*, but somewhat mitigated by the fact that different programmers who participate in the projects use coherent vocabularies to write code and documentation, thus increasing the chance that two words in different applications may describe the same requirement.

The sheer number of API calls suggests that many of these calls are likely to be shared by different programs that implement completely different requirements leading to significant imprecision in calculating similarities. We found that, for CLANdroid, over 95% of the apps in our dataset made use of the String object. Our dataset also shows that the `View` intent alone was instantiated 363,141 times, which is enough to appear in every app in our dataset 25 times.

If similarity scores are computed based on common API calls or intents such as these, most Android programs would be similar to one another. On top of that, it is not computationally feasible to compute similarity scores with high precision for hundreds of thousands of API calls. It is an instance of a problem known as *the curse of dimensionality*, which is a problem caused by the exponential increase in processing by adding extra dimensions to a representational space [63].

Graphically, programs are represented as dots in a multidimensional space where dimensions are semantic anchors and coordinates in this space reflect the numbers of these semantic anchors in programs. The JDK contains close to 115,000 API calls that are exported by a little more than 13,000 classes and interfaces that are contained in 721 packages. For CLANdroid, we note that the Android SDK encompasses over 3,500 classes that are contained in 200 packages. Furthermore, the Android SDK can also use the JDK. Computing similarity scores between programs using VSM in a space with hundreds of thousands of dimensions is not always computationally feasible, it is imprecise, and difficult to interpret. We need to reduce the dimensionality of this space while simultaneously revealing similarities between implemented latent high-level requirements.

2.3 Approach[10]

Our key idea is threefold. First, if two applications share some semantic anchors (e.g., API calls), then their similarity index should be higher than for applications that do not share any semantic anchors. Sharing semantic anchors means more than the exact syntactic match between the same two API calls; it also means that two different API calls will match semantically if they come from the same class. This idea is rooted in the fact that classes in JDK contain semantically related API calls; for example, the class `java.security.KeyStore` contains nested classes and API calls that enable programmers to implement requirements related to managing keys and certificates. For CLANdroid we must also consider the Android SDK. While classes such as the previously mentioned ones are also available in the Android SDK, one Android unique class would be `android.hardware.Sensor` which contains API calls to allow developers to provide support for hardware features, such as the phone's camera and other sensors. Another class is the `android.graphics.Canvas` class, which provides developers with the ability to display images or text on the screen. Thus, we exploit relationships between inheritance hierarchies in the JDK and Android SDK to improve the precision of computing similarity. This idea is related to semantic spaces where concepts are organized in structured layers and similarity scores between documents are computed using relations between layers [28]. Moreover, recent work has shown that API classes and packages can be used to categorize software applications using those classes and packages [48, 47, 49, 38].

Second, different API calls have different weights. Recall that many applications have

[10]Some of the material in this section was previously published in ICSE 2012 [45] and as a part of Dr. Collin McMillan's dissertation [44].

many API calls that deal with collections and string manipulations. Our idea is to automatically assign higher weights to API calls that are encountered in fewer applications and, conversely to assign lower weights to API calls that are encountered in a majority of applications. There is no need to know what API calls are used in applications -- this task should be done automatically. Doing it will improve the precision of our approach since API calls that come from common packages like `java.lang` will have less impact to skew the similarity index.

Finally, we observed that a requirement is often implemented using combinations of different API calls rather than a single API call; and in Android apps is often a combination of different API calls, intents, user permission declarations, and sensors usage. It means that co-occurrences of API calls in different applications form patterns of implementing different requirements. For example, a requirement of efficiently and securely exchanging XML data is often implemented using API calls that read XML data from a file, compress and encrypt it, and then send this data over the network. Even though different ways of implementing this requirement are possible, detecting patterns in co-occurrences of API calls and using these patterns to compute the similarity index may lead to higher precision when compared with competitive approaches.

Because *CLANdroid* uses not only API calls as semantic anchors but also Android intents, sensors, and permissions, and because Android intents are similar to APIs in that they are designed to enable the reuse of commonly sought functionality, we expect that the usage of Android intents as semantic anchors should follow the same ideas outlined above as API calls. That is, Android applications that share intents should have a higher similarity index than applications that do not share intents. While intents do not have different inheritance hierarchies as API calls do, different intents do have different weights. Thus, if all applications use an intent such as `ACTION_DIAL` (which displays the phone dialer with a provided number filled in), this intent would have a low weight, compared to a perhaps rarer intent like `ACTION_CREATE_DOCUMENT` (which allows the user to create a document). Also, similar to how combinations of API calls fulfill requirements as opposed to a single API call, combinations of intents are used to provide an Android application with all of its functionality, not a single intent.

2.3.1 Latent Semantic Indexing[11]

To implement our key idea we rely on an IR technique called *Latent Semantic Indexing (LSI)* that reduces the dimensionality of the similarity space while simultaneously revealing latent concepts that are implemented in the underlying corpus of documents [10]. In LSI, terms are elevated to an abstract space, and terms that are used in similar contexts are considered similar even if they are spelled differently. LSI automatically makes embedded concepts explicit using *Singular Value Decomposition (SVD)*, which is a form of factor analysis used to reduce dimensionality of the space to capture most essential semantic information.

The input to SVD is an $m \times n$ *term document matrix (TDM)*. Each of m rows corresponds to a unique term, which in our case is a class name that contains a corresponding API call that is invoked in a corresponding application (i.e., document). Columns correspond to unique documents, which in our case are Android mobile applications. Each element of the TDM contains the weight that shows how frequently this API call is used in this application when compared to its usage in other applications[12]. We cannot use a simple metric such as the API call count since it is biased -- it shows the number of times a given API call appears in applications, thus skewing the distribution of these calls toward large applications, which may have a higher API call count regardless of the actual importance of that API call.

SVD decomposes TDM into three matrices using a reduced number of dimensions, r, whose value is chosen experimentally. The number of dimensions for LSI is commonly chosen $r = 300$ [10, 62, 60]. One of these matrices contains document vectors that de-

[11]Some of the material in this section was previously published in ICSE 2012 [45] and as a part of Dr. Collin McMillan's dissertation [44].

[12]Note that we do not consider the number of times each API call is executed, e.g., in a loop. Instead, we count occurrences of API calls in source code.

scribe weights that documents (i.e., mobile apps) have for different dimensions. Each column in this matrix is a vector whose elements specify coordinates for a given application in the r--dimensional space. Computing similarities between applications means computing the cosines between vectors (i.e., rows) of this matrix.

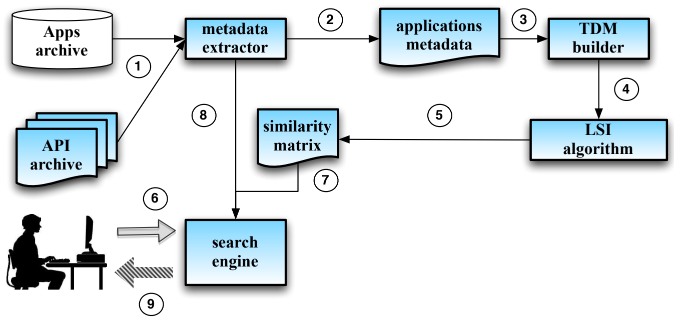


Figure 2.1: *CLANdroid* architecture and workflow.

2.3.2 *CLANdroid* Architecture and Workflow

The main elements for *CLANdroid* are the Android Applications, the App Decompiler, the Metadata Extractor (for extracting APIs, identifiers, etc.), the Term Document Matrix builder, and the LSI algorithm. In the TDM for *CLANdroid*, each row represents an application and and each column represents a different metadata value: API classes, identifiers, intents, permissions, or sensors. For *CLANdroid*, we only considered class-level similarities regarding API calls - we did not compute package-level similarities. While *CLAN* combines both package-level and class-level similarities and weights them evenly, we opted to compute only class-level similarities which utilize the full class name (e.g. `package.class`) because there are multiple packages in the Android SDK which have classes with the same name. For instance, both the packages `android.hardware` and `android.graphics` have a class called `Camera`. The class belonging to the former package allows the developer to utilize the camera built-in to the mobile device. However, the class belonging to the latter package allows the developer to compute 3D transformations and generate a matrix that could be used on a `Canvas`. By only using the class name (i.e. `Camera`) we introduce potential fuzz and mismatches into the data. By using the fully qualified class name we prevent this from occurring.

CLANdroid works as follows. First, all the apps must be obtained and downloaded - we downloaded 14,450 free Android apps from Google Play. We then had to decompile these apps, so that we may extract the different information used to compute similarities from the source code. Once each app had been decompiled to source code, we ran scripts to extract various data: identifiers, APIs, sensors, and intents from the source code, and permissions were extracted from the `AndroidManifest.xml` file that is in every app. For each application, we also extracted a goldset: this goldset was the list of apps that were displayed as similar apps from the app page on Google Play. We describe this data acquisition and extraction in detail in Section 2.4.

Once all the data is extracted, we first created a co-occurrence matrix which simply listed how many occurrences of a particular piece of unique metadata was found. For instance, the co-occurrence matrix for identifiers would have each row representing a document, and each column representing a unique identifier. The value would be the amount of times that unique identifier occurred in that particular application. Thus, we had five co-occurrence matrices in total: one for API calls at the class-level, one for identifiers, one for Android intents, one for permissions, and one for sensors.

We then converted these matrices to their TFIDF equivalents. We applied the LSI algorithm to each of these five TFIDF matrices and computed the cosine similarity between each application. The construction of the TDMs took anywhere between 20 minutes to 3 hours depending on the ranking method - the TDM generation for sensors was the fastest and the TDM generation for identifiers was the slowest. The running of SVD on these TDMs took anywhere between an hour to five hours, with the sensors TDM running the fastest and the identifiers TDM running the slowest. All of these computations were done on an Intel Xeon CPU X5672, 3.20 GHz with over 100 GB of RAM available. For each TDM, we found the following amount of unique metadata values: 981,945 identifiers, 469,552 APIs, 1,575 permissions, 309 intents, and 10 sensors.

2.3.3 Summary of CLAN study

McMillan *et al.* created an approach for detecting `Closely reLated ApplicatioNs (CLAN)` to help users detect similar related software applications given a Java application. *CLAN* was the first approach to use API calls as semantic anchors in order to find applications that functioned similarly. *CLAN* was used on 8,310 Java applications, and an experiment with 33 participants was conducted to evaluate the performance of *CLAN*. *CLAN* is compared against *MUDABlue*, which is the closest competitive approach. *MUDABlue* uses identifiers instead of API calls to compute similarities between applications. With strong statistical significance, *CLAN* has been shown to automatically detect similar applications with a higher precision than *MUDABlue*.

2.4 Finding Closely Related Android Applications

We conducted a study to determine how effective CLAN is when finding similar mobile apps, in particular Android apps. This study was driven by the following *goals*: (i) we wanted to evaluate whether the results of CLAN hold also in the context of Android apps (i.e., using APIs outperforms identifiers when using them for detecting similar applications); (ii) we wanted to evaluate other semantic anchors that are available for Android apps (i.e., intents, user permissions, and sensors) besides API calls; and (iii) because of the impact of third-party libraries and obfuscated code when using APK files in empirical studies [37], we also analyzed the impact of these two factors when detecting similar Android applications.

The *context* of the study is of 14,450 free Android applications that were downloaded from Google Play. The *quality focus* is the goldset of similar apps provided by Google Play[13] and the similarity of the apps as perceived by users. Besides the attributes eval-

[13] In addition to metadata and app store reviews, for a specific app, Google Play provides a list of similar apps in the same category.

uated with CLAN, we used other semantic anchors than can be extracted from Android apps (user permissions, intents, and sensors).

2.4.1 Study Design

To validate the accuracy of CLANdroid, in the context of this study we formulated the following research questions:

- RQ_1: *Does CLANdroid produce better results than MUDABlue?* This research question aims at validating if using APIs to detect similar applications outperforms identifiers as in the case of Java systems. Android apps are highly dependent on the Android SDK [57, 56, 52, 53]. Android apps use a considerable number API calls in common, and it is possible that API calls are not enough to identify variabilities across several apps and domain categories. Thus, we wanted to validate if the same results of CLAN in Java systems (i.e., API calls outperforms identifiers) hold on Android apps.
- RQ_2: *What semantic anchors used in CLANdroid produce better results when compared to the others?* The purpose of this research question is to explore other semantic anchor that are specific of Android apps such as user permissions declared in the manifest files, sensors used by the application, and Android intents. Specifically, we evaluated whether these Android-specific semantic anchors outperform API calls.
- RQ_3: *Do third-party libraries and obfuscated apps impact the accuracy of CLANdroid?*. Linares-Vásquez *et al.* [37] found that using APK files in empirical studies could introduce threats to the validity of the results because of the impact of third-party libraries and obfuscated code. In this study we used APK files to extract API calls, identifiers, user permissions, sensors, and intents. Thus, it is possible that

third-party libraries embedded in the APKs and obfuscated code impact the detection of similar apps.

The **independent variable** in the three research questions is the approach used for detecting similar apps ($CLANDroid_{API}, MUDABlue, CLANdroid_{Int}, CLANdroid_{Perm}, CLANdroid_{Sens}, Combined$). The **dependent variable** in RQ_1 is the similarity between a source app and a set of potentially-similar apps perceived by users. The **dependent variable** in RQ_2 and RQ_3 is the similarity ranking of the apps in the goldset when using a specific approach.

To analyze whether the results of CLAN also hold for CLANdroid (RQ_1), we designed a survey of 20 users aimed at comparing how similar are similar apps detected by CLANdroid (i.e., using API calls), a MUDABlue base approach (i.e., using identifiers), a Combined approach (i.e., API calls + identifiers), and the Google Play's goldset. In particular we asked participants to rank the similarity between a source app and a set of potentially similar apps by using the following Likert scale:

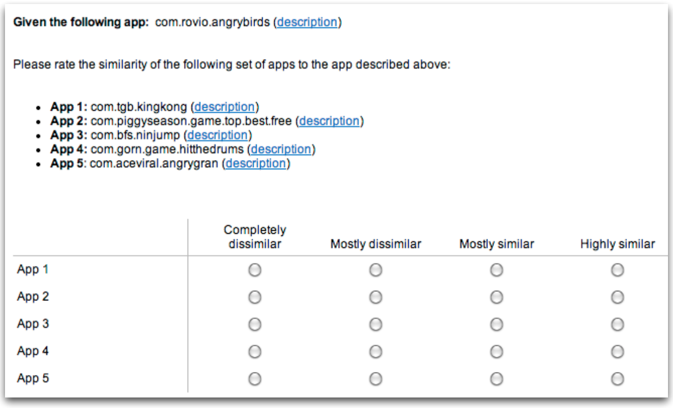
1. **Completely dissimilar:** The participant is highly confident that the app is dissimilar to the source app.
2. **Mostly dissimilar:** It is unclear if the app is similar to the source app.
3. **Mostly similar:** There are some similarities between the app and the source app.
4. **Highly similar:** The participant is highly confident that the app is similar to the source app.

For the survey we selected randomly 16 apps belonging to different domain categories (See Table 2.1), and for each app we built a pool of 4 sets of potentially-similar-apps; each set contains the top 5 similar apps detected by a specific approach (i.e., Goldset, CLANdroid, MUDABlue, Combined). Then, the survey was designed using the Qualtrics[14]

[14]`http://www.qualtrics.com/`

Table 2.1: Android apps used in the survey for RQ_1

App (category)	App (category)
com.rovio.angrybirds (Arcade)	air.BasketballDoodFree (Sports)
cn.wps.moffice_eng (Business)	cn.menue.barcodescanner (Tools)
com.virtual.guitar (Music & Audio)	com.adobe.psmobile (Photography)
com.e_gadget.MindFireF (Cards)	com.joey.video.player (Media & Video)
com.juandroidev.livecube (Personalization)	com.kangaroo.logic (Brain)
com.oanda.fxtrade (Finance)	com.officedepot.mobile.ui.bsd.us.prod (Shopping)
com.protecmedia.newsApp (News & Magazines)	com.rm.android.facewarp (Entertainment)
com.rs.autokiller (Productivity)	com.enlightenedapps.bubbleblaster (Casual)



Given the following app: com.rovio.angrybirds (description)

Please rate the similarity of the following set of apps to the app described above:

- **App 1:** com.tgb.kingkong (description)
- **App 2:** com.piggyseason.game.top.best.free (description)
- **App 3:** com.bfs.ninjump (description)
- **App 4:** com.gorn.game.hitthedrums (description)
- **App 5**: com.aceviral.angrygran (description)

	Completely dissimilar	Mostly dissimilar	Mostly similar	Highly similar
App 1	○	○	○	○
App 2	○	○	○	○
App 3	○	○	○	○
App 4	○	○	○	○
App 5	○	○	○	○

Figure 2.2: Example of survey's question for the app `com.rovio.angrybirdsspace .ads`

software in such a way that the participants had to answer 16 questions (i.e., one per app), and each question should have a set of similar apps selected randomly from the respective pool. For each app in the potentially-similar-set we asked the participants to rank the similarity to the source app using the Likert scale described before[15]. The package name of the apps (source and potentially-similar-set) and links to Google Play were provided with each question as in Figure 2.2. The results of the survey are presented and analyzed in Section 2.4.5.

To identify which approach (including Android-specific semantic anchors) produces better results (RQ_2), we used 14,450 free Android apps, and the list of similar apps listed

[15]The decision of 16 apps is based on the fact the we estimated that participants would spend no more than 3.5 minutes answering a question, and the time answering the survey should not be more than 1 hour.

by Google Play (i.e., goldset). Given an app $a_i \in A$, and A the context of our study, we used the five CLANdroid-based approaches (i.e., API calls, sensors, intents, user permissions, API calls+identifiers) and MUDABlue to detect similar apps to all the $a_i \in A$. The similar apps were detected in the complement set of each a_i (i.e., $A - a_i$). Instead of using a survey as in the case of CLAN and the RQ_1 of this study, we used the goldsets as a reference for evaluating the accuracy of the approaches. Our decision is motivated by the fact that comparing six different approaches requires a complicated survey design that requires a large number of participants and time; in addition the goldsets availability provided us with a ground truth for evaluating the approaches automatically. Therefore, after detecting similar apps for each a_i we looked for the ranking of the apps belonging to a_i goldset and evaluated the rankings using two metrics: the top rank (TOP_R) of any app in the goldset, and the average rank (AVG_r) of all the apps in the goldset. Given and app a_i, $a_j \in Goldset(a_i)$, $TOP_R(a_i)$ and $AVG_r(a_i)$ are computed as in Equations 2.1 and 2.2.

$$TOP_r(a_i) = \text{min}(rank(a_j)) \tag{2.1}$$

$$AVG_r(a_i) = \frac{1}{|Goldset(a_i)|} \sum_{j=1}^{|Goldset(a_i)|} rank(a_j) \tag{2.2}$$

For instance, given app X, if app X has apps A, B, and C in its goldset, we will check each of the five approaches to see the top rank (i.e., position closer to 1) of the apps A, B, C. Thus, app C may be detected at rank 20 for APIs, app A may be detected at rank 5 for identifiers, etc. For the average rank, we computed the average of the rankings for each app in the goldset (e.g., average of rank(A), rank(B), and rank(C) when using API calls). The results for RQ_2 are provided and analyzed in Section 2.4.5.

For RQ_3 we computed the top rank and average rank of the goldset as in RQ_2. However, we considered only project-specific classes (i.e., excluding third-party libraries), and we removed obfuscated apps. For considering only the project-specific code (and detecting obfuscated apps we followed the same procedure in [37]. The approach based on user permissions is not impacted by third-party libraries because user permissions are extracted from manifest files. The results and corresponding analysis for RQ_3 are presented in Section refsec-clandroid-results3.

2.4.2 Data Extraction

We downloaded 14,450 free mobile apps from Google Play as APK files, so that we may decompile them and extract information from the source code. When downloading an app, we also crawled goldset for the app (i.e., a set of apps listed by Google Play as similar) and domain category. Once we had the APK files, we converted them into `.java` source files using the following procedure: (i) unzip the APK files by using the *apktool*[16] tool, which reveals the compiled Android application code file (note that an APK is just a set of zipped DEX files); then (ii) translate the DEX files from the Dalvik bytecode to JAR files using the *dex2jar*[17] tool; then (iii) extract the `.class` files from the JAR files by using the *7zip*[18] tool; and finally (iv) decompile the `.class` files to `.java` files by using the *JAD*[19] decompiler tool (Figure 2.3).

In order to extract the amount of APIs used for each app, we used the *JClassInfo*[20] tool. To acquire the permissions used by each app, we extracted this information from the `AndroidManifest.xml` file that is present with every app. We extracted the `AndroidManifest.xml` file by also using the aforementioned *apktool*. The Android manifest file contains information such as the Java package for the application, which processes will host application components, and also which permissions the application must have so that it may access protected parts of the API. To obtain the intents from each app, we used a recursive `grep` command that pattern-matched only on intents that started with `new Intent("android.intent` and `.setAction("android.intent` so that we only

[16]http://code.google.com/p/android-apktool/
[17]http://code.google.com/p/dex2jar/
[18]http://www.7-zip.org/
[19]http://www.varaneckas.com/jad/
[20]http://jclassinfo.sourceforge.net/

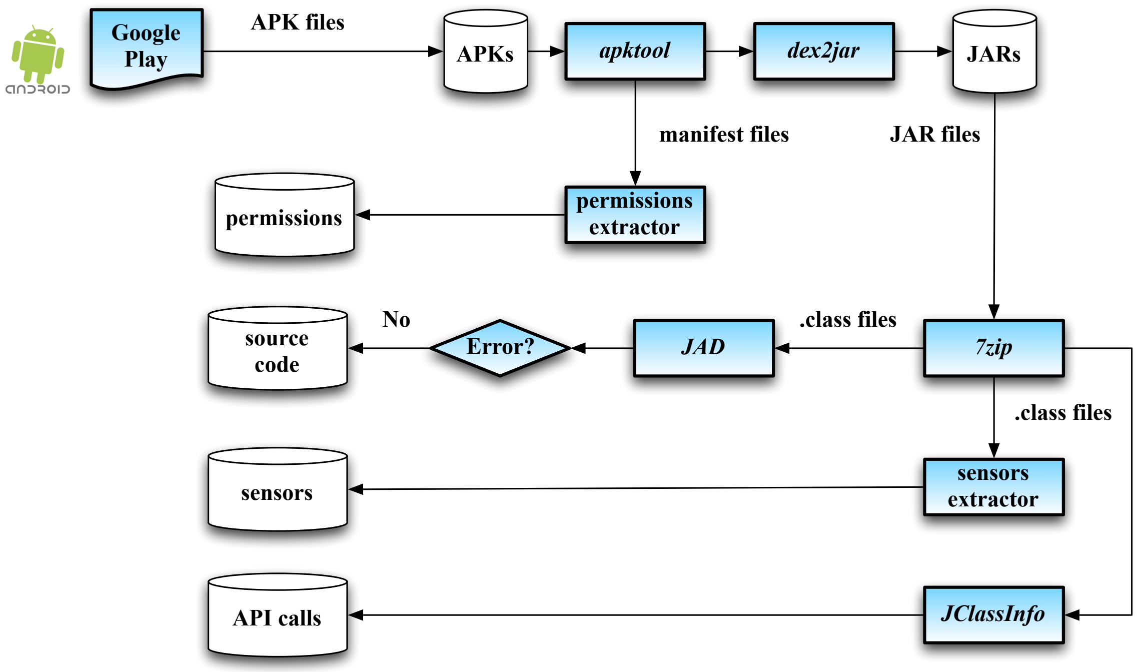


Figure 2.3: Data extraction process from APK files.

collected new instantiations of Android intents. By using this method, we included both implicit and explicit Android intents.[21] To extract the identifiers from the source code we applied a splitting pre-processing technique to this corpus. The splitting is done on underscores and on capital letters - all other non-literals are removed. So, the string `FileWriter out_writer` becomes the four identifiers `file writer out writer`. However, we did not use stemming or the removal of stop words, to be consistent with the design of CLAN.

To extract the sensors used by each app, we looked into the `.class` files extracted from the APKs (Figure 2.3). We searched for the string `"Landroid/hardware/SensorManager; ->getDefaultSensor(I)Landroid/hardware/Sensor;"`, which is the definition of the sensor manager. If this string is found, then we knew that the app is using sensors. Each instance of the string starts with `"invoke-virtual <Var1>, <Var2>"`, where `Var2` is a variable. For each instance of this string we found, we search above the line of code that

[21]An implicit Android intent only specifies the action to be performed, while an explicit Android intent specifies both the action to be performed and which component to perform it with.

Table 2.2: Number of apps per category

Category	#apps	Category	#apps
Arcade	636	Medical	50
Books and reference	315	Music and audio	639
Brain	830	News and magazines	574
Business	517	Personalization	909
Cards	463	Photography	454
Casual	689	Productivity	490
Comics	27	Racing	351
Communication	340	Shopping	137
Education	913	Social	172
Entertainment	1367	Sports	562
Finance	455	Sports games	394
Health and fitness	140	Tools	1101
Libraries and demo	100	Transportation	102
Lifestyle	890	Travel and local	345
Media and video	450	Weather	38

the instance was found for the declaration of this variable. One example of this variable declaration is this: `"const/4 v10, 0x1"` where `v10` is the variable and 0x1 is the value. We took this value and compare dit to the Google source code for Android[22] to detect which sensor is being used. We extracted the sensors information this way instead of from the manifest file from an app because it is not mandatory to list all used sensors in the manifest. The information in the manifest is only used to filter apps in Google Play based on those declared sensors.

Our distribution of apps by category is listed in Table 2.2. Thus, we used five types of attributes, with each type of attributing representing an approach. The five attributes are APIs, identifiers, intents, permissions, sensors, and represent the approaches we used for detecting similar Android apps: $CLANDroid_{API}, MUDABlue, CLANdroid_{Int}, CLANdroid_{Perm}, CLANdroid_{Sens}$. In addition, we included a combined approach, similarly to CLAN, which combines API calls and identifiers ($Combined$).

2.4.3 Analysis Method

Aimed at answering the research questions, we tested the following null hypotheses:

[22]https://android.googlesource.com/platform/frameworks/base/+/android-4.3_r2.1/core/java/android/hardware/Sensor.java

- H_{0_1}: *there is no significant difference in the values of similarity (S) per app between participants who use MUDABlue, CLANdroid, and Combined*
- H_{0_2}: *there is no significant difference in the values of precision (P_r) per app between participants who use MUDABlue, and CLANdroid.*
- H_{0_3}: *there is no difference in the average ranking of the goldset AVG_r when using MUDABlue, CLANdroid, and Combined.*
- H_{0_4}: *there is no difference in the top ranking of the goldset TOP_r when using MUDABlue, CLANdroid, and Combined.*
- H_{0_5}: *there is no difference between the AVG_r values collected for H_{0_3} and the AVG_r values collected when excluding third-party libraries from the context.*
- H_{0_6}: *there is no difference between the TOP_r values collected for H_{0_4} and the TOP_r values collected when excluding third-party libraries from the context.*
- H_{0_7}: *there is no difference between the AVG_r values collected for H_{0_3} and the AVG_r values collected when excluding obfuscated apps from the context.*
- H_{0_8}: *there is no difference between the TOP_r values collected for H_{0_4} and the TOP_r values collected when excluding obfuscated apps from the context.*

Hypotheses H_{0_1} and H_{0_2} were used to validate RQ_1. The survey results for RQ_1 were analyzed differently to CLAN's; CLAN tasks were designed with a reuse scenario in mind. In the case of CLANdroid we are assuming a more general scenario which includes users looking for substitutes or complementary apps[23]. Moreover, although some open source apps are distributed as APK files through Google Play, we were not interested in analyzing source code. Our context is of Android free apps distributed as APK files. Consequently, instead of measuring the confidence C we measured functional similarity as perceived by users that inspect Google Play. However, similarly to the CLAN study (Section 2.3.3), we

[23]End-users do not search/browse the source code of Android apps; they look for APK files

also computed P_r via the fraction of the top r ranked target applications that are relevant to the source application, where $r = 5$ in this experiment, which means that each similarity engine returned the top five similarity matches. We selected apps with at least 5 apps in their goldset that are in our dataset in order to represent each engine fairly, as there could be potential cases where only the top 2 apps are similar, or where the fourth ranked app is the only similar app. Along with the values of similarity S and precision P_r, we also examined the values of the first app returned by each set S_1 and the highest similarity ranking given to any app within the five apps returned by each set S_T.

To validate that the results of H_{0_1} and H_{0_2} are statistically significant we used the Kruskal-Wallis test [13]. Once we tested the null hypotheses $H_{0,1}$ and $H_{0,2}$, in case of accepting the alternative hypotheses, we followed a post-hoc test procedure aimed to compare the effectiveness of each approach when compared to other (e.g., MUDABlue vs $CLANdroid_{API}$, $CLANdroid_{API}$ vs Combined). We used the Mann-Whitney test [13] for pairwise comparisons.

Hypotheses H_{0_3} and H_{0_4} were used to validate RQ_2. In this case we followed the same procedure for H_{0_1} and H_{0_2}; all the approaches were compared initially using Kruskal-Wallis tests; then post-hoc test procedures were done for pairwise comparisons. Hypotheses H_{0_5} to H_{0_8} were used to validate RQ_3. For RQ_3, we only used pairwise comparisons without Bonferroni correction between the values of TOP_r and AVG_r collected for $H_{0,3}$/$H_{0,4}$ and the values collected for H_{0_5} to H_{0_8}. For example, to measure if there is an impact of third-party libraries when using $CLANdroid_{API}$, we compared the TOP_r values when using $CLANdroid_{API}$ on the study context, to the TOP_r values when using $CLANdroid_{API}$ and excluding third-party libraries from the context.

In all the tests we looked for statistical significance at an alpha level = 0.05 and because of the multiple comparisons we applied a Boferroni Correction to the p-values when required (i.e., hypotheses $H_{0_1} - H_{0_4}$). We also computed the Cliff's delta d effect size [25] to measure the magnitude of the difference in all the tests. We followed the guidelines in [25] to interpret the effect size values: negligible for $|d| < 0.147$, small for

$0.147 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$ and large for $|d| \geq 0.474$.). We are not assuming population normality and homogeneous variances, therefore we choose non-parametric methods (Kruskal-Wallis test, Mann-Whitney test, and Cliff`s delta).

2.4.4 Replication Package

The data set used in our study is publicly available at `http://www.cs.wm.edu/semeru/data/clandroid`. Specifically, we provide:(i) online implementation of CLANdroid; (ii) the list (and URLs) of the studied 14,450 free Android applications; (iii) the questions used for the survey; and (iv) the list of similar apps detected by each one of the analyzed approaches

2.4.5 Results

In this Section we present the results we got in this study aimed at answering the research questions in 2.4.1. We also include a list of cases we manually inspected to support our quantitative findings.

RQ_1: Do CLAN Results Hold on Android Apps?

The results are summarized in Figures 2.4 and 2.5, and Table 2.3. In our evaluation, the higher the values of precision P_r and functional similarity S are, the more effective a set was at displaying similar applications. When examining the average functional similarity for the four sets, Google Play was the highest with $S = 3.03$. This means that, on average, users ranked each app recommended by Google Play as **mostly similar**. MUDABlue followed next with $S = 2.52$. Although less than Google Play, this S value is still closest to **mostly similar**. The remaining two sets, $CLANdroid_{API}$ and Combined, returned $S = 2.24$ and $S = 2.15$ respectively. Therefore, users ranked each app on average from these sets as **mostly dissimilar**.

We then analyze the precision P_r, which is the fraction of retrieved instances that are similar. Essentially, the higher the value of P_r, the better the set of apps was at returning more apps that were more similar than dissimilar. Google Play's set had the highest average precision with $P_r = 0.705$, thus on average 70% of the apps returned by Google Play were considered more similar than dissimilar. Following Google Play again is MUDABlue, which resulted with an average precision of $P_r = 0.516$. Therefore, on average for MUDABlue, about 52% of the apps returned were deemed more similar than dissimilar by the users. Thus, both Google Play and MUDABlue returned more apps that were similar to the given app than dissimilar. $CLANdroid_{API}$ and the Combined sets, however, do not follow this pattern. They both contained lower precisions: $P_r = 0.409$ and $P_r = 0.393$, respectively. Thus we make an important observation: Google Play and MUDABlue were the only sets to return more apps that were ranked more similar than dissimilar ($P_r > 0.5$). We also note that the values of P_r and S share the same ranking of sets in terms of effectiveness: Google Play (best), MUDABlue, $CLANdroid_{API}$, and Combined (worst).

However, we also investigated two other metrics in order to further understand the results: how well the first app performed in terms of similarity ranking (which we denote S_1), and how well the most similar app out of the $r = 5$ (as ranked by users) performed on average in terms of similarity ranking (which we denote S_T). The ranking of sets when examining S_1 remained unchanged from the previous results of P_r and S. However, all similarity values increased, which is likely a result of taking into consideration only the top ranked app by each engine and Google Play. The S_1 values for Google Play, MUDABlue, $CLANdroid_{API}$, and Combined are $S_1 = 3.204$, $S_1 = 2.971$, $S_1 = 2.898$, and $S_1 = 2.866$. Although the ranking of sets remains unchanged, this brings to light an important observation: when taking into consideration only the first app returned by each engine or Google Play, on average, all apps are ranked **mostly similar**. However, when we take into account the similarity ranking S_T for the most similar app on average out of the five apps displayed by each set, the ranking of sets changes slightly. The

Table 2.3: Results of the user survey. The first column indicates what is being measured (e.g., S for functional similarity), and the second column indicates which set of similar apps were used (i.e., similar apps detected with a specific approach). The next 16 columns represent the 16 questions presented to a user in the survey, and the last column contains the average value of each row. The rows show the average values from all users that answered a question for a specific set of similar apps

Var	Approach	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Avg
S	Goldset	2.142	2.17	3.332	3.178	3.572	3.136	2.67	3.652	3.67	2.64	3.228	1.8	3.5	3.134	3.15	3.546	3.033
S	$CLANdroid_{API}$	2.5	2.65	2.228	1.9	1.84	1.878	2.45	2.5	3.778	1.95	1.934	2.058	2.858	1.75	1.7	1.866	2.24
S	MUDABlue	1.9	2.92	2.266	1.334	2.268	3.45	2.232	3.198	3.67	2.57	2.56	2.95	3.45	1.234	1.954	2.398	2.522
S	Combined	1.6	2.8	2.2	1	2.04	2.12	2.4	2.4	3.134	2.05	2.35	3.05	2.598	1.314	1.532	1.734	2.145
P_r	Goldset	0.4	0.2	0.833	0.822	0.857	0.8	0.5	0.925	1	0.68	0.75	0.2	1	0.733	0.65	0.927	0.705
P_r	$CLANdroid_{API}$	0.54	0.65	0.429	0.233	0.2	0.3	0.5	0.55	1	0.25	0.267	0.4	0.629	0.2	0.2	0.2	0.409
P_r	MUDABlue	0.3	0.64	0.4667	0.133	0.2	0.9	0.4	0.933	0.833	0.571	0.68	0.65	0.75	0	0.333	0.467	0.516
P_r	Combined	0.2	0.6	0.3	0	0.28	0.44	0.6	0.52	0.8	0.3	0.35	0.8	0.6	0.029	0.2	0.267	0.393
S_1	Goldset	2.86	1.71	3.5	3.78	4	3.67	2.67	3.75	3.67	3.2	3.88	2.25	3.5	2.67	3.25	2.91	3.204
S_1	$CLANdroid_{API}$	4	3.5	4	1.33	4	2.13	2.5	2	3.75	2.5	2	2.29	2.86	2.25	3.25	4	2.898
S_1	MUDABlue	3.5	3.8	3.33	1.67	3.67	2	2.83	3.33	3.67	2.43	2.2	3.25	3.25	1.5	3.11	4	2.971
S_1	Combined	3	4	4	1	4	2	3.25	1.8	2.67	2	2.5	3.5	3.71	1.43	3.33	3.67	2.866
S_T	Goldset	2.86	4	3.5	3.78	4	3.67	2.67	3.88	3.67	3.2	4	2.25	3.5	4	3.5	3.91	3.524
S_T	$CLANdroid_{API}$	4	3.5	4	4	4	2.63	3.25	4	3.88	2.5	2	2.75	3.29	2.25	3.25	4	3.331
S_T	MUDABlue	3.5	3.8	3.33	1.67	3.67	4	2.83	3.33	3.67	3	3.2	3.25	4	1.5	3.11	4	3.241
S_T	Combined	3	4	4	1	4	2.8	3.25	2.8	3.67	2.75	2.5	3.75	3.71	1.71	3.33	3.67	3.121

set for Google Play returned with the similarity value $S_T = 3.524$. Thus, we notice that when taking into account this value, that Google Play, on average, will return at least one app out of the five in the set that is marked **highly similar**. The next ranked set is $CLANdroid_{API}$, which has interchanged positions with MUDAblue. $CLANdroid_{API}$ has the similarity value $S_T = 3.331$, and therefore on average will return at least one out of its top five ranked apps that is **mostly similar**. MUDABlue and the Combined sets return with the similarity values $S_T = 3.24$ and $S_T = 3.12$ respectively, and thereby are similar to $CLANdroid_{API}$ with the result that they will on average return at least one out their top five ranked apps that will be deemed **mostly similar**.

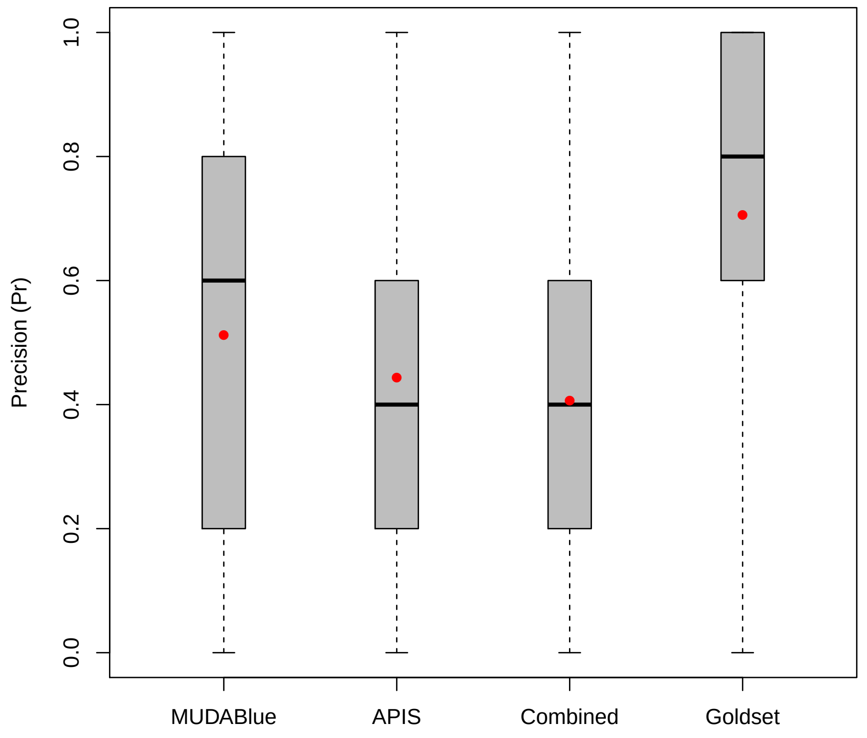


Figure 2.4: Survey results: distribution of the precision (Pr) depicted by approach. Each boxplot was plotted using all the responses from the participants for the similar-apps set generated with the considered approaches.

Summarizing, the results of our **RQ**$_1$ show that *the results of CLAN do not hold on Android mobile apps. We found that when looking at both the average functional similarity of the top five apps returned by each set and the average precision of the top five apps returned by each set, the Google Play set returns the highest values and thus outperforms the other sets. MUDABlue outperforms the remaining sets* $CLANdroid_{API}$ *and Combined, of which* $CLANdroid_{API}$ *outperforms Combined. These results are also mirrored when measuring the average reported functional similarity of the top returned app from each set. However, when measuring the top average reported functional similarity of the five returned apps,* $CLANdroid_{API}$ *outperforms MUDABlue.*

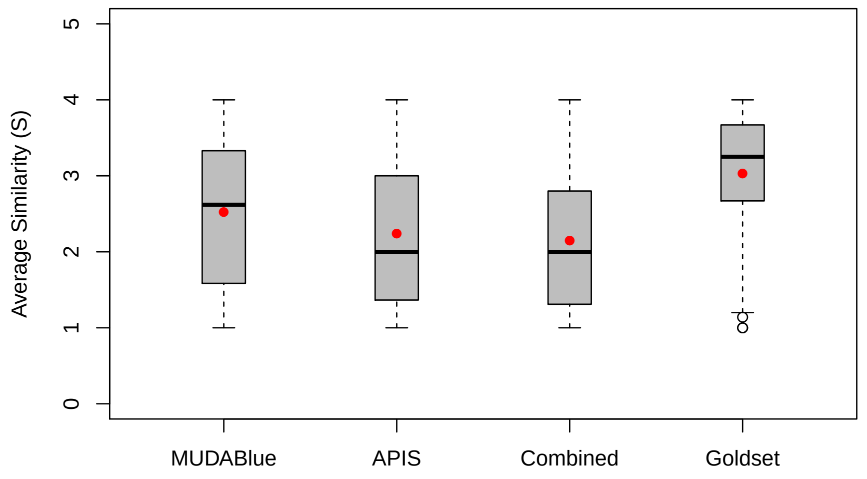


Figure 2.5: Survey results: distribution of the average similarity (S) depicted by approach. Each boxplot was plotted using all the responses from the participants for the similar-apps set generated with the considered approaches.

RQ_2: Accuracy of CLANdroid Semantic Anchors

The results are summarized in Figures 2.6 and 2.7, and Table 2.4. The figures depict the distribution of the rankings provided by the different methods to the apps in the goldset; the table lists the results of the statistical tests for the hypotheses H_{0_3} and H_{0_4}.

On average, using APIs and Intents delivers the worst rankings when analyzing TOP_r. The mean values in Figure 2.6 in ascending order are: 858.2 for $CLANdroid_{Sens}$ (median 1), 1996 for MUDABlue (median 974.5), 2143 for $CLANdroid_{Perm}$ (median 940), 2183 for Combined (median 1128), 2524 for $CLANdroid_{Int}$ (median 1404), and 2638 for $CLANdroid_{API}$ (median 1763). This result is also reflected in the case of AVG_r (Figure 2.7). From best AVG_r to worst, based on the average value in the boxplot, we get: $CLANdroid_{Sens}$ (mean=3424, median=2531), $MUDABlue$ (mean = 4844, median =4733), $CLANdroid_{Perm}$ (mean =4948, median = 4747), Combined (mean=5165, me-

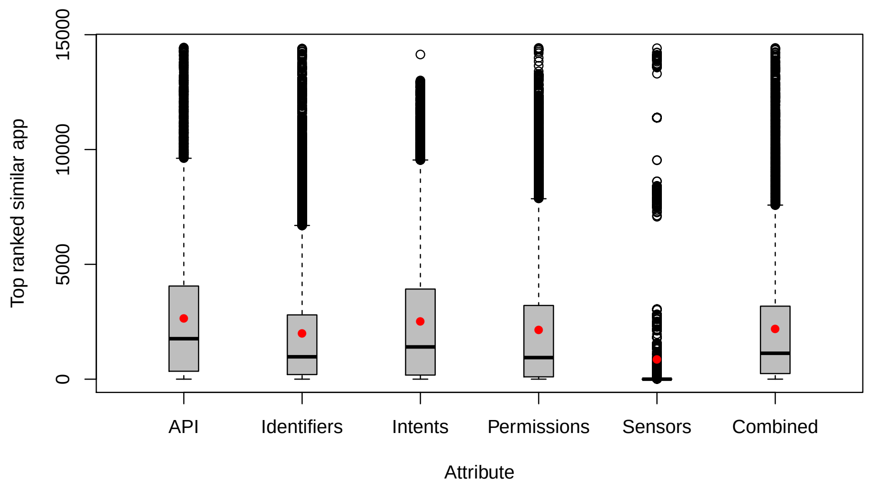


Figure 2.6: Boxplots by ranking method, measured by the top ranked app in goldset.

dian = 5075), $CLANdroid_{Int}$ (mean = 5597, median = 5467), and finally $CLANdroid_{API}$ (mean = 5837, median = 5818).

Regarding the null hypotheses (H_{0_3} and H_{0_4}), we found that the p-value for the Kruskal-Wallis is less than 0.05 when using both metrics (which means that there are significant differences between the rankings when analyzing AVG_r and TOP_r. The post-hoc procedures with the Mann-Whitney confirm the initial results, showing statistical significant difference in all the cases (Table 2.4) except for the comparison between MUDABlue and $CLANdroid_{Perm}$, and $CLANdroid_{Int}$ and Combined. However, when looking into the magnitude of the differences, (i.e., Cliff's delta) in most of the comparisons the differences are negligible (i.e., $|d| < 0.147$) and small (i.e., $0.147 \leq |d| < 0.33$). The magnitudes are only medium and large when comparing the results of using Sensors as semantic anchors against the others; this case is confirmed with the boxplots, which show that the best rankings are provided when using Sensors (i.e., $CLANdroid_{Sens}$).

The values suggest that $CLANdroid_{Sens}$ is the best approach, followed by $MUDABlue$ and $CLANdroid_{Perm}$. However, $CLANdroid_{Sens}$ only appears to be the best approach

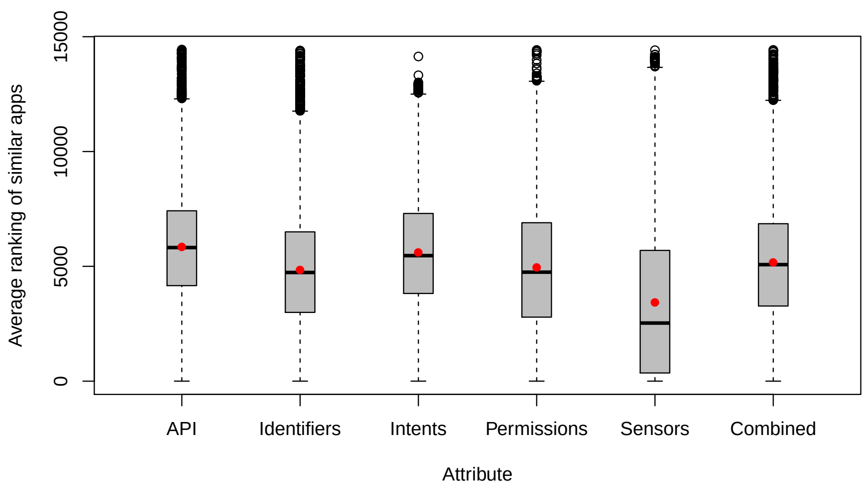


Figure 2.7: Boxplots by ranking method, measured by the average ranked app in goldset.

because sensors are not widely used in our dataset, and there are only 13 types of sensors than can be used by Android apps[24]. CLANdroid ranks applications with the exact same similarity values at the same rank, therefore, the sensors attribute may not be as useful for detecting similar apps. For instance, if app A has three apps in its goldset (apps B, C, and D), and apps B and C both utilize the exact same sensors as app A, then they will both have a similarity value of 1.0 when compared to app A. Thus, both apps will be ranked at position 1, and app D will be ranked at position 3, as it is the third-most similar app to app A.

This is most noticeable when finding apps that are similar based on the phone sensors used by the application, due to both the low number of unique sensors and that sensors function as a boolean value. Due to the low number of unique sensors used in our dataset (10), it can be common for apps to use the same combination of sensors, especially if the app only uses one or two sensors. This also means that all apps that do not use any phone sensors have a perfect similarity value as well. 11,385 of the apps in our dataset

[24]`http://developer.android.com/guide/topics/sensors/sensors_overview.html`

Table 2.4: Results of statistical tests for H_{0_3} and H_{0_4} when using the whole dataset.With the Bonferroni correction the new alpha value for the post-hoc tests is 0.005 (0.05/10).

Var	Approach 1	Approach 2	P-value post-hoc	W-statistic	Cliff's delta
AVG_r	$CLANdroid_{API}$	MUDABlue	0.0000	100438216.0	0.2647
AVG_r	$CLANdroid_{API}$	$CLANdroid_{Int}$	0.0000	86803527.5	0.0712
AVG_r	$CLANdroid_{API}$	$CLANdroid_{Perm}$	0.0000	97981639.5	0.2288
AVG_r	$CLANdroid_{API}$	$CLANdroid_{Sens}$	0.0000	121952415.0	0.4820
AVG_r	$CLANDROID_{API}$	Combined	0.0000	94502214.0	0.1856
AVG_r	MUDABlue	$CLANdroid_{Int}$	0.0000	68309631.5	-0.2600
AVG_r	MUDABlue	$CLANdroid_{Perm}$	0.1080	80944464.5	-0.0306
AVG_r	MUDABlue	$CLANdroid_{Sens}$	0.0000	109780235.0	0.3189
AVG_r	MUDABlue	Combined	0.0000	76145791.0	-0.1023
AVG_r	$CLANdroid_{Int}$	$CLANdroid_{Perm}$	0.0000	93580421.0	0.1716
AVG_r	$CLANdroid_{Int}$	$CLANdroid_{Sens}$	0.0000	118726949.0	0.4445
AVG_r	$CLANDROID_{Int}$	Combined	0.0000	89644271.0	0.1227
AVG_r	$CLANdroid_{Perm}$	$CLANdroid_{Sens}$	0.0000	109068917.0	0.3259
AVG_r	$CLANDROID_{Perm}$	Combined	0.0000	77524323.5	-0.0643
AVG_r	$CLANDROID_{Sens}$	Combined	0.0000	50314473.0	-0.5945
TOP_r	$CLANdroid_{API}$	MUDABlue	0.0000	94197008.5	0.1739
TOP_r	$CLANdroid_{API}$	$CLANdroid_{Int}$	0.0000	88554034.5	0.0310
TOP_r	$CLANdroid_{API}$	$CLANdroid_{Perm}$	0.0000	95336872.5	0.1316
TOP_r	$CLANdroid_{API}$	$CLANdroid_{Sens}$	0.0000	144147643.5	0.4008
TOP_r	$CLANDROID_{API}$	Combined	0.0000	90559271	0.1245
TOP_r	MUDABlue	$CLANdroid_{Int}$	0.0000	77848027.5	-0.1637
TOP_r	MUDABlue	$CLANdroid_{Perm}$	0.0000	84608427.5	-0.0449
TOP_r	MUDABlue	$CLANdroid_{Sens}$	0.0000	142883538.5	0.2892
TOP_r	MUDABlue	Combined	0.0000	78135817.5	-0.0588
TOP_r	$CLANdroid_{Int}$	$CLANdroid_{Perm}$	0.0000	87974000.0	0.0998
TOP_r	$CLANdroid_{Int}$	$CLANdroid_{Sens}$	0.0000	135735336.5	0.3709
TOP_r	$CLANDROID_{Int}$	Combined	0.1796	82687393.5	0.0917
TOP_r	$CLANdroid_{Perm}$	$CLANdroid_{Sens}$	0.0000	134783209.0	0.3091
TOP_r	$CLANDROID_{Perm}$	Combined	0.0000	75896398.5	-0.0116
TOP_r	$CLANDROID_{Sens}$	Combined	0.0000	20171738.5	-0.4775

make no use of any of the sensors, and thus all of these apps are deemed similar when ranked by sensors alone. However, while the sensors ranking method alone may not be the most effective, it can be combined with other ranking methods to help detect similar applications more accurately.

Without considering $CLANdroid_{Sens}$, detecting the goldset apps as similar by using identifiers (i.e., MUDABlue) appears to be the best approach, as it consistently has a lower average and lower median when compared to the other methods. Ranking by permissions is second best, beating out both APIs and intents. Although there is a list of 145 official

Android permissions[25], we detected over 10 times this amount of unique permissions in our dataset. This is possible due to apps being able to create custom permissions, such as `com.motorola.launcher.permission.READ_SETTINGS`. This permission is a part of the StartApp SDK[26], which is a third-party SDK that "contains code necessary to have `out of App' monetization channels for your application."

Permissions are a unique way of detecting similar apps due to its wide variance in ranking apps. Thus, the more permissions an app has, the likelier the top ranked apps by CLANdroid are functionally similar. The opposite also holds true: if an app has a single permission such as `android.permission.INTERNET`, then every app which has only this permission will be marked with a perfect similarity. For this reason, we recommend using permissions in conjunction with another measurement attribute, but not using permissions alone.

One example that demonstrates the ineffectiveness of permissions is when considering the app *Slots Royale - Slot Machines*[27]. This app has four Android standard permissions: `READ_PHONE_STATE`, `ACCESS_COARSE_LOCATION`, `ACCESS_NETWORK_STATE`, and `INTERNET`. The app *Tennis Score*[28] has these exact same permissions, and thus is marked as a perfectly similar app in $CLANdroid_{Perm}$ (and because it is perfectly similar, it must be at rank 1). However, the app *Slots Free (5 Slot Machines)*[29] is a similar app part of the goldset, and while it contains the four permissions that *Slots Royale* has, it also has an additional five different permissions. Simply adding these additional permissions pushes the similarity ranking of this app down from a perfect similarity (rank 1) to rank 1,550.

On average, rankings of the goldset apps are far from the top-positions in all the approaches. When using TOP_r, there were only 471 apps in our dataset that had an app in their goldset ranked at position 1 for any ranking method (e.g., app A may have an app from its goldset ranked at position 1 for identifiers, while app B may have an app from its

[25]http://developer.android.com/reference/android/Manifest.permission.html
[26]http://developers.startapp.com/Resource/SDK/Startapp%20SDK%20integration%20manualV1.5.pdf
[27]https://play.google.com/store/apps/details?id=com.mw.slotsroyale
[28]https://play.google.com/store/apps/details?id=RobotMoose.TennisScore
[29]https://play.google.com/store/apps/details?id=com.viaden.slotsfree

goldset ranked at position 1 for sensors). When using TOP_r but only taking into consideration apps from our dataset that are of the same category as the queried app, this number increases to 1,134. This shows that at least 663 apps had an app of a different category being ranked higher than an app in the queried app's goldset for any ranking method.

To find an explanation to this we manually inspected the results. One explanation is that the goldsets only include apps in the same category, however, *CLANdroid* detects similar applications across different categories. Also, we found evidence of apps ranked by *CLANdroid* at top positions, which do not belong to the goldset, but are still closely related. For example, when we checked the rankings for the popular game app *Angry Birds*[30], the top ranked app for each approach was *Angry Birds Space*[31]. The second ranked app by APIs and identifiers was *Amazing Alex Free*[32], which is also developed by Rovio. The second ranked app by intents was *Hamster: Attack!*[33] (by Backflip Studios), an app in the `Casual` category. The third ranked app by APIs and identifiers is also the same, with the app being *The Sims FreePlay*[34]. The apps in the goldset do not appear to be functionally similar to *Angry Birds*, where the goldset contains apps such as *Angry Monkey*[35] and *NinJump*[36].

Another example are the apps *Home Architecture and Design*[37] and *The Social Business*[38]; both apps were likely developed using the AppMakr tool according to their package name. AppMakr is a "what you see is what you get" editor that allows users to build apps with no coding knowledge[39]. Although these apps belong to different categories (`Lifestyle` and `Business` respectively) and authors, they make the exact same API calls and have almost the exact same identifiers. The intents, permissions, and sensors used

[30]https://play.google.com/store/apps/details?id=com.rovio.angrybirds
[31]https://play.google.com/store/apps/details?id=com.rovio.angrybirdsspace.ads
[32]https://play.google.com/store/apps/details?id=com.rovio.amazingalex.trial
[33]https://play.google.com/store/apps/details?id=com.backflipstudios.android.hamsterattack
[34]https://play.google.com/store/apps/details?id=com.ea.games.simsfreeplay_na
[35]https://play.google.com/store/apps/details?id=com.tgb.kingkong
[36]https://play.google.com/store/apps/details?id=com.bfs.ninjump
[37]https://play.google.com/store/apps/details?id=com.appmakr.app346687
[38]https://play.google.com/store/apps/details?id=com.appmakr.app166847
[39]http://www.appmakr.com/

are also identical. However, while these apps are ranked similarly, there are also instances of AppMakr apps which don't rank similar to apps created the same way. For instance, comparing the *Home Architecture and Design* app to *All Design*[40](an `Education` app), all of the rankings are at least 80 or higher, except for the intents ranking attribute, where the rank is 11. In fact, out of the 10 AppMakr-based apps in our dataset, when using one as a query app for *CLANdroid*, the other nine always rank within the top 20 similar apps based on intents. This may demonstrate that although certain apps created via the same program (such as AppMakr) may have different API calls and identifiers, they may still be very similar when compared based on intents.

A representative app from the `Books` category is called *The Bible,The Qur'an & Science*[41]. This app is simply a book discussing religion and science. Google Play has the app *The Holy Quran - English*[42] listed as a related app. When using *CLANdroid* considering only apps in the `Books` category, this app ranked in positions 76, 163, 104, 201, and 1 (APIs, identifiers, intents, permissions, and sensors respectively) in regards to the former app. However, when we look at the top ranked apps by *CLANdroid*, another app (*Islam in Brief*[43]) by the same developer ranks at number 1 in both APIs and identifiers (with APIs having a perfect similarity ranking), while the other three ranking attributes have multiple apps ranked at number 1. Thus, we observe that there are apps in which both appear to have similar content, but are implemented differently. Furthermore, we recognize that apps from the same developer are ranked highly similar, analogous to the scenario between *Angry Birds* and *Amazing Alex*.

Upon querying the Finance app *Money Notes Lite*[44], we found that the top 3 apps ranked by identifiers not only belong to the Finance category, but all help the user manage expenses. These three apps are *T2Expense - Money Manager*[45], *Home Budget with*

[40]https://play.google.com/store/apps/details?id=com.appmakr.app120673
[41]https://play.google.com/store/apps/details?id=com.smartersoft.smarterbooks.en.book8
[42]https://play.google.com/store/apps/details?id=com.verypositive.Quran
[43]https://play.google.com/store/apps/details?id=com.smartersoft.smarterbooks.en.book7
[44]https://play.google.com/store/apps/details?id=app.moneynoteslite
[45]https://play.google.com/store/apps/details?id=com.t2.t2expense

Sync Lite[46], and *HardCash Tracker*[47]. Although none of these apps appear in the goldset for *Money Notes Lite*, the apps appear to blend in well with the goldset, thus demonstrating that they are similar. For this app, the apps ranked by the semantic anchors varied wildly in functionality. The top app by $CLANdroid_{API}$ is a `News & Magazines` app and is dissimilar from the queried app. The top results for $CLANdroid_{Combined}$ seem to be somewhat related to the queried app as they are financial calculators (e.g. *CalcPack Financial Calculators*[48]), but are not completely related in the functionality that they provide. The remaining three semantic anchors have at least 60 apps marked with a perfect similarity ranking (thus are at rank 1), and they also vary largely in functionality from apps that utilize the camera to games. This example not only demonstrates the effectiveness of identifiers when detecting similar apps, but also that when either $CLANdroid_{API}$ or MUDABlue detect apps that are not functionally similar, combining them can help attempt to detect similar apps that are more relevant than just one engine alone, as seen with *CalcPack Financial Calculators*.

Another interesting example comes from an app called *Babanev Kereso Fiu*[49] and the app *Babanev Kereso Lany*[50] in its goldset. These are both Hungarian apps which were designed to help parents choose a name for their child by providing information about the name such as the origin, meaning, and any other information. When running *CLANdroid* on *Babanev Kereso Fiu*, we found that the goldset app *Babanev Kereso Lany* is ranked number one in all the approaches, and has a perfect similarity score in every ranking method except identifiers. This is to be expected, as both apps function exactly the same, except with the former having information on male names and the latter having information on female names, which would change the identifiers in the source code. This example shows how similar apps from the same developer use both similar semantic anchors and also identifiers, although the identifiers are not a perfect match (i.e., the similarity score is

[46]https://play.google.com/store/apps/details?id=com.anishu.homebudget.lite
[47]https://play.google.com/store/apps/details?id=de.maxmaurer.hardcashtracker
[48]https://play.google.com/store/apps/details?id=pack.calc.calcpack
[49]https://play.google.com/store/apps/details?id=com.origo.babyname
[50]https://play.google.com/store/apps/details?id=com.origo.babynamegirl

less than 1.0).

We queried the app *250+ Solitaire Collection*[51] which is an app that contains 253 different solitaire games for a user to play. After inspecting the apps in its goldset, we found *Solitaire Free Pack*[52] as the most similar to *250+ Solitaire Collection*. We believe this app to be the most similar as it contains multiple forms of solitaire for the user to play. Although this app ranked at positions 543 and 14 for APIs and identifiers respectively, it ranked at the top position (i.e., rank 1) when combining these two measurement attributes. Thus, we observe that combining identifiers and API calls can improve the results when using only API calls.

Summarizing, the results of our **RQ**$_2$ show that *the while the semantic anchors used in $CLANdroid_{Sens}$ and $CLANdroid_{Perm}$ appear to detect similar apps better than the other semantic anchors, this is not the case. Due to the way these anchors function as binary values (either they are present or not), they create many false positives as they match with a large quantity of apps. Between the remaining two semantic anchors $CLANdroid_{API}$ and $CLANdroid_{Int}$, we conclude that the intent semantic anchor slightly outperforms API calls when used for detecting similar Android apps.*

RQ_3 On the Impact of Third Party Libraries and Obfuscated Apps

The results are summarized in Figures 2.8 and 2.9, and Table 2.5. The figures depict the distribution of the rankings provided by the different methods to the apps in the goldset when excluding the third-party libraries from the analysis; the table lists the results of the statistical tests for the hypotheses H_{0_3} and H_{0_4}. We did not include the results of $CLANdroid_{Perm}$ because user permissions were extracted from manifest files, therefore, excluding third-party libraries from the analysis does not change the results.

In terms of AVG_r, again, MUDABlue is the best approach and $CLANdroid_{API}$ the

[51]https://play.google.com/store/apps/details?id=com.anoshenko.android.solitaires
[52]https://play.google.com/store/apps/details?id=com.tesseractmobile.solitairefreepack

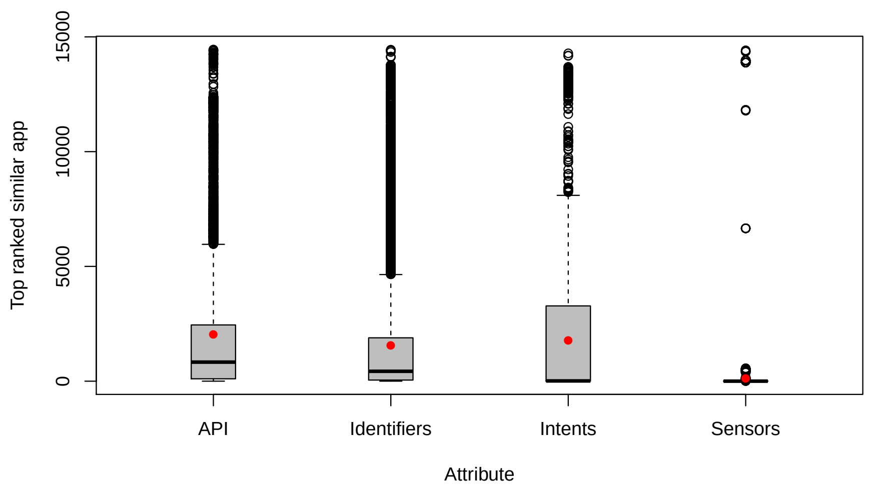


Figure 2.8: Boxplots by ranking method, measured by the top ranked app in goldset when excluding third-party libraries.

worst: $CLANdroid_{Sens}$ (mean = 1058, median = 1), MUDABle (mean = 4350 , median = 3991), $CLANdroid_{Int}$ (mean = 4740, median = 4860), $CLANdroid_{API}$ (mean = 5253, median = 5040); in terms of TOP_r the results also hold: $CLANdroid_{Sens}$ (mean = 120.4, median = 1), MUDABle (mean = 1569 , median = 430.5), $CLANdroid_{Int}$ (mean = 1775, median = 11), $CLANdroid_{API}$ (mean = 2044, median = 830.5). However, there are significant differences between the values of TOP_r and AVG_r produced by the CLANDroid approaches when including and excluding third-party libraries. Table 2.5 lists the differences in the means and medians of TOP_r and AVG_r after excluding third-party libraries from the analysis. The negative differences (columns Diff(mean) and Diff(average)) confirm that on average, the ranking of the goldset apps were improved.

The app *Night Vision Cam*[53] experiences a large increase in its average and top goldset rankings when removing third-party libraries from the data. One app in the goldset for *Night Vision Cam* is *LiveKey Camera*[54]. When using the whole dataset including third-

[53]https://play.google.com/store/apps/details?id=androix.com.android.NightVisionCam
[54]https://play.google.com/store/apps/details?id=com.sonyericsson.androidapp.appkey

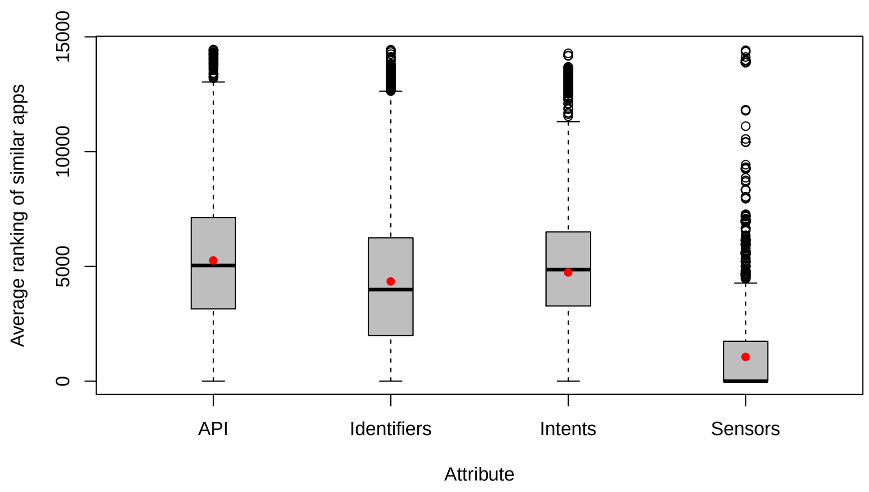


Figure 2.9: Boxplots by ranking method, measured by the average ranked app in goldset when excluding third-party libraries.

Table 2.5: Differences between TOP_r and AVG_r when including and excluding third-party libraries. Negative values in the Diff(mean) and Diff(median) columns shows that when excluding third-party libraries the rankings move in the direction of the top positions.

Var	Approach	Diff (mean)	Diff (median)	P-value Man-Whitney	W-statistic	Cliff's delta
TOP_r	$CLANDROID_{API}$	-594.0856	-932.5000	0	97812782.5	0.1532
TOP_r	MUDABlue	-426.7505	-544.0000	0	96957040.5	0.1243
TOP_r	$CLANDROID_{Int}$	-749.5034	-1393.0000	0	106958259.5	0.1878
TOP_r	$CLANDROID_{Sens}$	-737.8482	0.0000	0	93014599.5	0.2414
AVG_r	$CLANDROID_{API}$	-584.3896	-777.7333	0	93204770.5	0.1561
AVG_r	MUDABlue	-494.0487	-741.6250	0	92403260	0.1319
AVG_r	$CLANDROID_{Int}$	-857.6693	-606.4805	0	96283253.5	0.2400
AVG_r	$CLANDROID_{Sens}$	-2365.6368	-2530.0000	0	122227310.0	0.5011

party libraries, the similarity rankings of *Night Vision Cam* were 4928 for $CLANdroid_{API}$ and 6852 for MUDABlue. However, when excluding third-party libraries, these rankings improved tremendously to 50 and 266 respectively, with the other attributes also reflecting this change. Upon further investigation, we found that while *LiveKey Camera* contained only app-specific code (i.e., no third-party libraries), *Night Vision Cam* utilized Google Ads. We observed that *Night Vision Cam* had only 21 classes in its main package, whereas the Google Ads library had 156 classes. This example demonstrates the large impact that TPLs can have when detecting similar apps, particularly in cases such as this where the

Table 2.6: Number of obfuscated apps per category

Category	#apps	Category	#apps
Arcade	49	Medical	0
Books and reference	30	Music and audio	67
Brain	108	News and magazines	33
Business	45	Personalization	191
Cards	42	Photography	79
Casual	115	Productivity	86
Comics	0	Racing	44
Communication	41	Shopping	9
Education	54	Social	8
Entertainment	68	Sports	28
Finance	57	Sports games	44
Health and fitness	4	Tools	149
Libraries and demo	5	Transportation	4
Lifestyle	42	Travel and local	8
Media and video	44	Weather	4

TPL "outweighs" the app-specific code due to size and amount of classes.

Table 2.7 summarizes the results of the comparison of TOP_r and AVG_r when including and excluding obfuscated apps. There are significant differences, but the magnitudes of the differences are negligible, in most of the cases, according to the Cliff's delta. The negligible values, in terms of effect size, could be explained by the fact that only 1,458 obfuscated apps were identified in our dataset (Table 2.6 lists the distribution of obfuscated apps per category).

In the case of AVG_r, the columns Diff(mean) and Diff(median) show an improvement in the average rating of the goldset apps. This is not a surprising result because removing the obfuscated apps reduces the size of the dataset, and obfuscated apps ranked in top positions are removed from the ranking list. Therefore, AVG_r results after removing obfuscated apps suggest that *CLANdroid* is able to find similar apps even including obfuscated apps because API calls, sensors, user permissions, and Intents are part of the Android SDK (i.e., their calls/declarations in Android apps can not be obfuscated). There is an interesting example between the apps *Flip Clock NicePink Widget 4x2*[55] and *FlipClock NiceAll Pink Widget*[56]. While at first glance these two apps appear extremely similar except for a small graphical change, the latter app actually contains obfuscated

[55]https://play.google.com/store/apps/details?id=factory.widgets.FlipClockNicePink
[56]https://play.google.com/store/apps/details?id=factory.widgets.FlipClockNiceAllPink

code inside its main package. Furthermore, both of these apps contain the app *Live Wallpaper Clock Trial*[57] in their goldset. When examining how well this app ranks compared to both the obfuscated and non-obfuscated app, the non-obfuscated one has this app at ranking 9 for identifiers, while the former has the app ranked at position 65. Thus, we see a direct observation of how obfuscation can impact the similarity rankings between an obfuscated app and a non-obfuscated app when using identifiers.

However, the behavior of TOP_r is different. The values in Table 2.5 for TOP_r are positive, which means that the top ranked apps from the goldset, on average, lost positions in the ranking list. This behavior is not evident when analyzing the AVG_r, and is opposite to the improvement of AVG_r. Upon manual inspection of the apps, we found a possible explanation for the case of TOP_r when excluding obfuscated apps. For instance, in our dataset we have four goldset apps for the *Smart AppLock (App Protector)*[58] app. However, when we removed obfuscated apps there were only two goldset apps remaining as the other two were marked as obfuscated apps. We noticed that in MUDABlue for *Smart AppLock*, an obfuscated app (*AppLock*[59]) is ranked at the top position of 134, while the next highest ranked goldset app known as *App Lock (Smart App Protector)*[60] is at position 7220. However, after removing obfuscated apps from the dataset, *App Lock (Smart App Protector)* becomes the top-ranked app with its position changing to 4460.

We noticed a similar case when detecting similar apps to *Infinite Racing*[61]. We found four apps in our dataset which are in the goldset, and when we removed the obfuscated apps we had two goldset apps remaining (thus there were two obfuscated apps and non-obfuscated apps in the original goldset for this app). Similarly to the previous example, the ranking of the two non-obfuscated apps improved after the removal of obfuscated apps. However, the new ranking of the non-obfuscated apps is not as good as the obfuscated-apps ranking.

[57]https://play.google.com/store/apps/details?id=sgolovanov.GSFlipClockWallpaperTrial
[58]https://play.google.com/store/apps/details?id=com.thinkyeah.smartlockfree
[59]https://play.google.com/store/apps/details?id=com.domobile.applock
[60]https://play.google.com/store/apps/details?id=com.sp.protector.free
[61]https://play.google.com/store/apps/details?id=com.Sailfish.InfinityRacing

Table 2.7: Differences between TOP_r and AVG_r when excluding obfuscated apps. Negative values in the Diff(mean) and Diff(median) columns shows that when excluding obfuscated apps the rankings move in the direction of the top positions.

Var	Approach	Diff (mean)	Diff (median)	P-value Man-Whitney	W-statistic	Cliff's delta
TOP_r	$CLANDROID_{API}$	204.6992	261.0000	0.0000	67904601	-0.0589
TOP_r	MUDABlue	170.5645	161.5000	0.0000	67767769	-0.0542
TOP_r	$CLANDROID_{Int}$	15.4483	234.0000	0	68528932	-0.0043
TOP_r	$CLANDROID_{Perm}$	193.0679	131.0000	0.0000	66996707	-0.0565
TOP_r	$CLANDROID_{Sens}$	123.8036	0.0000	0.0000	69275605.5	-0.0360
AVG_r	$CLANDROID_{API}$	-427.9498	-464.0333	0.0000	77995659	0.1246
AVG_r	MUDABlue	-316.2017	-319.2500	0.0000	75946584	0.0923
AVG_r	$CLANDROID_{Int}$	-884.3345	-1028.9881	0.0000	84753555.5	0.2462
AVG_r	$CLANDROID_{Perm}$	-240.3167	-116.7857	0.0000	73796354.5	0.0661
AVG_r	$CLANDROID_{Sens}$	-370.2851	-229.2000	0.0000	76106276	0.0872

Summarizing, the results of our **RQ**$_3$ show that *the accuracy of CLANdroid is significantly impacted by the inclusion of third-party libraries. Excluding third-party libraries moved the average rankings (AVG_r) up by, on average, a minimum of 490 positions. We found that without including third-party libraries, each engine improved both its top app rankings (TOP_r) and average rankings significantly. However, while we also found that there are differences in the rankings when excluding apps we detected as obfuscated, the magnitudes of these differences are negligible in most cases. We also found that while the average rankings improved when we removed the obfuscated apps, the top rankings worsened due to obfuscated apps occupying the top ranks.*

2.4.6 Threats to Validity

In this section, we discuss threats to the validity of the experimental design for CLANdroid and how we address and minimize these threats.

Internal Validity

Goldsets. It is important to note that similarity between applications in the goldsets for CLANdroid are not symmetrical. Thus, if app B is found in the goldset for app A, this does not mean that app A will be in the goldset for app B. Because we reduce the goldset so that it only contains apps we have in our dataset (for practical reasons we cannot continuously download goldset apps), this means that app B may have no apps in our dataset that are also in its goldset. However, although app B may not have relevant goldset apps to be ranked, we cannot discard app B from the dataset as any apps that have app B in their dataset will then suffer, or even worse, remove the last app in their own goldset, thus turning them into an "outlier" app like app B.

We also assume that the goldsets provided by Google Play are "perfect." That is, each app listed as similar is most definitely similar. However, upon observation this may not be the case. If one goes to the page for an app on Google Play, if the app is a popular app and has many downloads, the apps listed similar are very likely to also be somewhat popular. Yet, if one were to navigate to the page for a more obscure app (for instance `C-Marbles 1 [falls]`[62]), it appears that the similar apps are selected based on identifiers in the app's name. So, just because an app in the goldset might have a low ranking in our similarity scores, it may be because the apps above it are actually more similar and relevant.

Application Categories. Regarding our similarity rankings based on the category of the app used as a query, we must remember that the similarity score is already affected by the other apps in the dataset even if they are of a different category. That is, even if we only rank apps that are of the same category as the app queried, due to the way TFIDF is computed, the TDM for LSI is affected, and thus the similarity scores are changed. However, if we decided to run LSI for each app only using the apps from each category, we would have to run both the TDM Builder and LSI computations 155 times (five times regardless of category for the different ranking methods, and 150 times for those five times for each category).

Main Package Extraction. When we extracted the main package from the manifests of each app in order to compute similarities between apps without including any information from third-party libraries (TPLs), we found that some apps did not specify a main package in their manifest. In these cases, we chose to use the name of the app's package

[62]https://play.google.com/store/apps/details?id=info.ryuojima.android.cfalls

as the main package, as the majority of the apps in our dataset followed this design. However, we also detected 649 apps that specified a main package in their manifest that did not exist in the decompiled source code. For instance, the app Race, Stunt, Fight, Lite![63] has the package name `ac.lite`, thus the non-TPL information should be decompiled to the `/ac/lite/` directory, but this directory does not exist within this app.

We investigated this by examining the first activity to be launched in the manifest (the first class to be executed), which we detected by searching for the `Launcher` Android intent also within the manifest. For this app, we found that the first class to be executed was `com.unity3d.player.UnityPlayerProxyActivity`. Unity3D[64] is a game engine that can be used to generate cross-platform apps, thus we know that this app used this game engine to generate some of the code for this app. However, due to the use of a third-party engine such as Unity3D, we are unable to distinguish parts of classes or even entire classes that were created solely by the developers. Some of the other 649 apps use various libraries and engines, such as MonoGame[65] or even the developers own library used for multiple apps. However, because these classes and code weren't written specifically for a single app, we opted to exclude this code to prevent high similarities between apps simply because they relied on the same library or engine. Thus, we were unable to extract any information from the source code of these 649 apps due to the inability to distinguish what code did and did not belong to a TPL because the specified main package did not exist.

External Validity

Application Dataset. Although we downloaded many apps for our experiment, we cannot guarantee these results to be the same for the entirety of Google Play. However, the apps we downloaded covered all the domain categories, and thus we believe that this dataset

[63]https://play.google.com/store/apps/details?id=ac.lite
[64]https://unity3d.com/
[65]http://www.monogame.net/

Table 2.8: Recent studies of similar app detection, purpose of study, and information used in the study. For platform we use M for mobile and D for desktop. The next column lists the number of apps in the dataset, and the TPL column marks if the study considered the impact of third-party libraries with a YES, NO, or NA (not applicable). Finally, the market category states where the apps were acquired from- MM : multiple markets, NR : not reported, GP : Google Play, FB : FreeBSD, SF : SourceForge, E: Eclipse Plugins.

Study	Purpose	Information Type	Platform	#apps	TPL	Market
Michail and Notkin [51]	Detecting similar libraries	Library source code	D	NA	NA	NR
Kawaguchi *et al.* [32]	Automatic Categorization	Source code identifiers	D	41	NA	SF
Crussell *et al.* [7]	Detecting cloned and rebranded apps	Java bytecode	M	$>$265K	YES	MM
Li *et al.* [35]	Using similarities to address security	File directories	M	$>$58K	NO	MM
Bajracharya *et al.* [2]	Source code retrieval	API calls from source	D	346	NA	E
Chen *et al.* [5]	Detecting cloned apps to address security	Methods from SMALI code	M	$>$150K	YES	MM
Cubranic *et al.* [76]	Recommending Software Artifacts	Issue-tracking	D	1	NA	E
Moritz *et al.* [58]	API search engine	API methods	D	13K	NA	NR
Gorla *et al.* [21]	Finding unadvertised behavior in apps	API invocations from SMALI	M	$>$22K	YES	GP
Desnos *et al.* [12]	Detection of similar apps	Custom method signatures	M	2	NO	GP
Ye *et al.* [77]	Context-aware Browsing	Component repository	D	NR	NA	NR
McMillan *et al.* [48, 47, 49]	Finding relevant functions	Function call graph	D	$>$ 18K	NA	FB
Thung *et al.* [74]	Detecting similar applications	Collaborative tagging	D	$>$100K	NA	SF

is a sufficient representative of Google Play.

2.5 Related Work

Several recent papers have used various methods in an attempt to accurately detect similar software and Android applications. The motivations for this have varied from detecting clones to prevent the plagiarism of app developers, to finding apps injected with malware. Other recent papers have utilized different techniques to creating helpful code source search engines. Table 2.8 summarizes related work which we describe in the following paragraphs.

Michail and Notkin [51] presented the tool CodeWeb, which is an automated approach for comparing and contrasting software libraries based on matching similar classes and functions cross libraries (via name and similarity matching). This work was inspirational for us in extending the relevance framework with semantic anchors. In contrast to CodeWeb, CLANdroid also uses advanced dimensionality reduction techniques based on LSI and SVD and computes similarities among applications in the context of the complete software repository. Kawaguchi *et al.* [32] created MUDABlue, which is also closely related to both CLANdroid and CodeWeb. MUDABlue is the closest competitor to CLANdroid, and as mentioned in Section 2.2.3, uses syntagmatic associations in order to compute similarities between applications.

Bajracharya *et al.* [2] developed a technique known as Structural Semantic Indexing (SSI), which was used to retrieve API usage examples from source code repositories. SSI was designed to be an effective retrieval scheme which used no documents other than source code. Bajracharya *et al.* measured three different forms of usage-based similarity: term frequency-inverse document frequency (TF-IDF) and two measurements which used feature vectors - Hamming Distance and Tanimoto Coefficient. Finally, the authors presented a technique to dynamically generate API usage snippets to provide helpful information to developers. CLAN and CLANdroid are different from SSI for three reasons: 1) CLAN/CLANdroid locate the applications similar to a given application, and does not require a natural-language query, 2) CLAN/CLANdroid are independent of the keywords chosen in the code, and 3) CLAN/CLANdroid have been evaluated using a standard methodology with programmers against a state-of-the-art approach (MUDABlue).

Crussell *et al.* [7] created the tool AnDarwin, which is a scalable approach for detecting similar Android apps using semantic information. AnDarwin uses the methods of an app to create a semantic block, and then creates a semantic vector to represent each of these semantic blocks. If two semantic blocks are code clones, then the semantic vectors representing these blocks will be similar. AnDarwin uses multiple techniques to attain its efficiency, such as Locality Sensitive Hashing (which allows the efficient detection of approximate near-neighbors in large quantities of vectors) and grouping the vectors based on their magnitudes (which improves scalability). AnDarwin was able to detect almost 4.3K apps cloned out of over 265K apps in their dataset, which was acquired through downloading apps from multiple markets along with the official Google Play market.

Cubranic *et al.* [76] designed the tool Hipikat that forms an implicit memory from the information stored in a project's archives, and then recommends artifacts from the archives that are relevant to the task that the developer is trying to perform. Hipikat is formed from two parts: the first being the group memory, and the second being the artifacts presented to the developer. There are four types of artifacts represented in the implicit group memory: bug and feature descriptions, source file revisions, messages posted on forums (e.g., mailing lists), and other project documents. Thus, Hipikat implements three distinct functions: identifying artifacts as they are added to a project's history, selecting relevant artifacts in response to a query, and updating the implicit group memory to reflect additions and changes to the project's archives.

Another similar paper is by Ye *et al.* [77] which proposes the software agent CodeBroker. CodeBroker is designed to automatically locate and present a list of software components that could be potentially useful for a developer during the current development situation. CodeBroker consists of three subsystems: the Listener, the Fetcher, and the Presenter. The Listener is constantly running and formulates queries by monitoring the activities of the software developer. The Fetcher finds and retrieves matching components from these queries. Finally, the Presenter displays the retrieved components that it deems related based on the background knowledge of the targeted developer. The components are retrieved from a large component repository. Thus, the authors present the idea of context-aware browsing to help present a list of contextualized components to developers without requiring direct operations from them.

Li *et al.* [35] created the tool DStruct, which is used to determine similarity among Android apps by utilizing the directory structures of the apps' archive formats. After extracting the APK, DStruct walks through the directories and files of the app to construct a tree which represents the directory structure. In this tree, the leaves represent files and non-leaves are directories. Note that this tree isn't walking through the decompiled source code and files, but instead just the extracted APK - which includes files such as

`AndroidManifest.xml`, `classes.dex`, and the `META-INF` directory. Li computes the percent difference between two trees to represent the similarity between two applications. Thus, the smaller the percent difference the more similar the apps are based on their directory structures. DStruct was used to find 3 instances of piracy and 9 instances of known malware from a dataset of 58,000 applications downloaded from Google Play and the Anzhi market[66].

Chen *et al.* [5] used a characteristic of geometry known as a centroid to achieve both accuracy and scalability in the detection of cloned apps. This centroid is created from dependency graphs and is used to measure the similarity between methods of two apps. However, these similarity measures are used to draw a boolean value conclusion on the app's core functionality cloning. That is, either two apps are marked to be clones or are not clones, which prevents partial similarity detection. Chen *et al.* evaluated their approach across mutliple different Android markets, but did not use Google Play.

Mortiz *et al.* [58] created the interactive code search tool ExPort. ExPort allows a user to select API methods as queries, to which the search engine will return usage examples related to the task. The authors use Relational Topic Modeling to compute similarities between APIs. Thus, when the user selects an API relevant to his or her task, the relevant APIs are shown in a call graph, and the call graph displays other functions that call the APIs. The user may then select functions from this call graph to further investigate API usage examples. ExPort used 13K open source Java systems in its dataset, and future work will include a user study to evaluate how effective the tool is.

Gorla *et al.* [21] created CHABADA, which is a tool that aims to accurately detect if an app does what it claims to do. Topic modeling using Latent Dirichlet Allocation is used on the descriptions of over 32K apps. The K-means algorithm is used to cluster the apps and thus provide the authors with the ability to identify groups of applications with similar descriptions. Once the apps are clustered based on the similarity of their description topics, the APIs for each app are extracted from the APK. The authors then choose to

[66]http://www.anzhi.com/

select a subset of sensitive APIs which are governed by an Android permission setting. Finally, the CHABADA uses a support vector machine to identify API outliers.

Desnos [12] used method signatures to detect similar Android apps, where the signatures were composed of string literals, API calls, control flow structures, and exceptions. However, this approach was only applied to two apps, so it is unknown if the approach would work for a larger dataset. Thung *et al.* [74] used an approach based on CLAN to detect similar software applications, but instead of using API calls the authors used the tags for applications in SourceForge[67]. However, this requires applications to be tagged properly, and thus applications without tags or tagged improperly will be tough to find similar applications for. Nonetheless, this approach could be used on the descriptions of apps in Google Play, and even potentially combine similarity rankings between descriptions and other attributes for measurements (such as APIs or identifiers).

McMillan *et al.* [48, 47, 49] created a code search system known as Portfolio, which is designed to assist programmers in finding definitions of functions. Portfolio uses the idea of a call graph, with functions as nodes and the directed edges which specify usages of these functions. By combining natural language processing and indexing techniques with a modified PageRank algorithm as well as a modified spreading activation network, Portfolio is able to assist programmers in the reusing of retrieved code as functional abstractions.

2.6 Summary

We created a search system for finding closely related Android applications (*CLANdroid*) that helps users find similar or related mobile apps. Our main contribution is an expansion upon the novel approach of CLAN by using features unique to Android applications as semantic anchors, and computing similarity scores between these Android applications. We extracted similar apps for our dataset from Google Play and performed two measures of

[67]http://sourceforge.net/

effectiveness on each of five different attributes. We also conducted a survey of 20 users who ranked the similarity of returned apps of different sets. The results show that when using a dataset that spans all domain categories in Google Play, identifiers are the most effective attribute at ranking apps in the goldset provided by Google Play. The survey results show that while Google Play suggests more similar apps than when compared to $CLANdroid$ and other engines, the engine using identifiers (MUDABlue) outperforms $CLANdroid$. This is likely due to the fact that while Android apps have differing functionality, they all must perform certain similar actions such as drawing on the screen or displaying text to the user, and these things are all part of the Android SDK which therefore makes identifiers more prominent. In the certain cases where the app returned by $CLANdroid_{API}$ is the most similar or ranked the best, it is likely that the both the app queried and returned utilize a rare API call.

Chapter 3

Conclusion

In conclusion, this thesis is composed of two major studies: the first being an examination of the impact of third-party libraries and obfuscated apps on the detection of class reuse and class cloning, and the second being a search engine which utilizes an Android mobile application as a query in order to return similar mobile Android applications. Thus, we have studied and enabled reuse in Android mobile applications.

In Chapter 1, we challenged the validity of previous empirical studies that have analyzed various factors within a dataset built of Android applications. We analyzed these threats to validity by investigating the impact of both third-party libraries and code obfuscation practices when estimating the amount of reuse by class cloning. We found statistically signifianct results that the inclusion and exclusion of third-party libraries impacts the amount of class cloning detected, and also found that the inclusion and exclusion of obfuscated apps impacts the amount of class cloning and can introduce false positives into the cloned signatures detection. Finally, we provided two actionable guidelines for future researchers: the first being that researchers should expect the amount of clone detection to be inflated if third-party libraries are included in the dataset, and the second being that researchers should expect the amount of clone detection to be inflated if obfuscated apps are included in the dataset.

In Chapter 2, we created $CLANdroid$, an extension to the approach known as CLAN, in which we used various different attributes related to Android applications in order to detect similar apps. We considered API calls, Android intents, permissions declared, and sensors usage as semantic anchors, and compared the results of these search engine to MUDABlue - a competitive approach that uses identifiers instead of semantic anchors to detect similar applications. We evaluated $CLANdroid$ using both goldsets obtained from Google Play and a conducted survey with 20 users. We found that while Google Play returned more apps that were deemed similar to users than the other search engines, MUDABlue consistently outperformed the $CLANdroid$ engine which used APIs in the survey. We also found that MUDABlue consistently had higher rankings for the apps in the goldset as opposed to any of the $CLANdroid$ engines which used semantic anchors. We recognize that this likely occurs because of the Android programming model - all apps must perform actions like displaying text to the user and drawing on the screen. As these actions are all part of the Android SDK, we recognize that identifiers then become more prominent when detecting similar Android applications.

Bibliography

[1] Nicolas Anquetil and Timothy C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON*, page 4, 1998.

[2] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *FSE*, pages 157--166, 2010.

[3] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assigment problem. *Commun. ACM*, 37(5):72--82, 1994.

[4] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine using free-form queries. In *FASE*, pages 385--400, 2009.

[5] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE'14*, 2014.

[6] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, 1997.

[7] Jonathan Crussell, Clint Gibler, and Hao Chen. Scalable semantics-based detection of similar android applications. In *ESORICS'13*, 2013.

[8] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage determining the provenance of software development artifacts. *Empirical Software Engineering*, 2012.

[9] J. Davies, D. M. German, M. W. Godfrey, and A. J. Hindle. Software bertillonage: Finding the provenance of an entity. In *MSR'11*, 2011.

[10] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391--407, 1990.

[11] Uri Dekel and James D. Herbsleb. Improving api documentation usability with knowledge pushing. In *ICSE*, 2009.

[12] A. Desnos. Android : Static analysis using similarity distance. In *45th Hawaii International Conference on System Sciences*, pages 5394--5403, 2012.

[13] Sheskin D.J. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & All, 2007.

[14] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964--971, 1987.

[15] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 147--156, New York, NY, USA, 2010. ACM.

[16] Google. Building and running, available at `http://developer.android.com/tools/building/index.html`.

[17] Google. Intent api reference guide, available at `http://developer.android.com/reference/android/content/Intent.html`.

[18] Google. Security and design, available at `http://developer.android.com/google/play/billing/billing_best_practices.html`.

[19] Google. Sensors overview, available at `http://developer.android.com/guide/topics/sensors/sensors_overview.html`.

[20] Google. System permissions, available at `http://developer.android.com/guide/topics/security/permissions.html`.

[21] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *ICSE'14*, 2014.

[22] Mark Grechanik, Kevin M. Conroy, and Katharina Probst. Finding relevant applications for prototyping. In *MSR*, page 12, 2007.

[23] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. Exemplar: Executable examples archive. In *Proc. of 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pages 259--262, 2010.

[24] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad M. Cumby. A search engine for finding highly relevant applications. In *ICSE (1)*, pages 475--484, 2010.

[25] R.J. Grissom and J.J. Kim. *Effect sizes for research: Univariate and multivariate applications*. Taylor & Francis, New York, NY, 2012.

[26] Mark. Harman, Yue. Jia, and Yuanyuan Zhang. App store mining and analysis: Msr for app stores. In *MSR'12*, pages 108--112, 2012.

[27] James Howison and Kevin Crowston. The perils and pitfalls of mining Sourceforge. In *MSR*, 2004.

[28] X. Hu, Z. Cai, A. C. Graesser, and M. Ventura. Similarity between semantic spaces. In *CogSci'05*, 2005.

[29] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. SpringerVerlag, 2004.

[30] Capers Jones. *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., 3rd edition, 2008.

[31] S. Kawaguchi, P. Garg, M. Matsushita, and K. Inoue. MUDABlue: An automatic categorization system for open source repositories. *International Journal of Systems and Software*, 79(7):939--953, 2006.

[32] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. Mudablue: an automatic categorization system for open source repositories. *J. Syst. Softw.*, 79(7):939--953, 2006.

[33] Kostas Kontogiannis. Program representation and behavioural matching for localizing similar code fragments. In *CASCON '93*, pages 194--205. IBM Press, 1993.

[34] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131--183, 1992.

[35] Saung Li, Steve Hanna, Ling Huang, Edward Wu, Charles Chen, and Dawn Song. Juxtapp and dstruct: Detection of similarity among android applications. Technical Report UCB/EECS-2012-111, Department of Computer Science, The University of Auckland, 2012.

[36] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *ESEC/FSE'13*, pages 477--487, 2013.

[37] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *11th IEEE Working Conference on Mining Software Repositories (MSR'14)*, page to appear, 2014.

[38] Mario Linares-Vásquez, Collin McMillan, and Denys Poshyvanyk. On using machine learning to automatically classify software applications into domain categories. *Empirical Software Engineering (EMSE)*, 2012.

[39] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *KDD'06*, pages 872--881. ACM Press, 2006.

[40] Wei Liu, Ke-Qing He, Jiang Wang, and Rong Peng. Heavyweight semantic inducement for requirement elicitation and analysis. *Semantics, Knowledge and Grid*, 0:206--211, 2007.

[41] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *ICSM'13*, 2013.

[42] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher. Recommending source code for use in rapid software prototypes. In *34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*, pages 848--858, 2012.

[43] C. McMillan, M. Linares-Vásquez, D. Poshyvanyk, and M. Grechanik. Categorizing software applications for maintenance. In *ICSM'11*, pages 343--352, 2011.

[44] Collin McMillan. *Searching, Selecting, and Synthesizing Source Code Components*. Doctoral dissertation, 2012.

[45] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting similar software applications. In *ICSE'12*, pages 364--374, 2012.

[46] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering (TSE)*, 38(5):1069--1087, 2012.

[47] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: A search engine for finding functions and their usages. In *Proc. of*

33rd IEEE/ACM International Conference on Software Engineering (ICSE'11), pages 1043--1045, 2011.

[48] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding relevant functions and their usages. In *ICSE'11*, 2011.

[49] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4), 2013.

[50] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. Recommending source code examples via api call usages and documentation. In *Proc. of 2nd ICSE2010 International Workshop on Recommendation Systems for Software Engineering (RSSE'10)*, 2010.

[51] Amir Michail and David Notkin. Assessing software libraries by browsing similar classes, functions and relationships. In *ICSE '99*, pages 463--472, New York, NY, USA, 1999. ACM.

[52] Roberto Minelli. *Software Analytics for Mobile Applications*. Master's thesis, 2012.

[53] Roberto Minelli and Michelle Lanza. Software analytics for mobile applications: Insights and lessons learned. In *CSMR*, 2013.

[54] Stefano Mizzaro. Relevance: The whole history. *JASIS*, 48(9):810--832, 1997.

[55] Stefano Mizzaro. How many relevances in information retrieval? *Interacting with Computers*, 10(3):303--320, 1998.

[56] Israel Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed Hassan. A large scale empirical study on software reuse in mobile apps. *IEEE Software Special Issue on Next Generation Mobile Computing*, 2013.

[57] I.J. Mojica Ruiz, M. Nagappan, B. Adams, and A.E. Hassan. Understanding reuse in the Android market. In *ICPC'12*, pages 113--122, 2012.

[58] Evan Moritz, Mario Linares-Vaśquez, Denys Poshyvanyk, Mark Grechanik, Collin McMillan, and Malcom Gethers. Export: Detecting and visualizing api usages in large source code repositories. In *ASE'13*, 2013.

[59] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *ESEC/FSE'13*. ACM, August 2013.

[60] D. Poshyvanyk and D. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proc. of 15th IEEE ICPC*, pages 37--48, Banff, Alberta, Canada, 2007. IEEE CS Press.

[61] Denys Poshyvanyk and Mark Grechanik. Creating and evolving software by searching, selecting and synthesizing relevant source code. In *Proceedings of the 31st IEEE/ACM International Conference on Software Engineering (ICSE 2009)*, pages 283--286, 2009.

[62] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420--432, 2007.

[63] Warren B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2007.

[64] Reinhard Rapp. The computation of word associations: comparing syntagmatic and paradigmatic approaches. In *19th ICCL*, pages 1--7, Morristown, NJ, USA, 2002.

[65] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar java classes using tree algorithms. In *MSR '06*, pages 65--71, New York, NY, USA, 2006. ACM.

[66] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[67] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P.G. Bringas. On the automatic categorization of Android applications. In *CCNC*, pages 149--153, 2012.

[68] David Schuler, Valentin Dallmeier, and Christian Lindig. A dynamic birthmark for java. In *ASE '07*, pages 274--283.

[69] S. Schulze and D. Meyer. On the robustness of clone detection to code obfuscation. In *IWSC*, pages 62--68, 2013.

[70] A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying Android applications using machine learning. In *CIS*, pages 329--333, 2010.

[71] Jeffrey Stylos and Brad A. Myers. A web-search tool for finding API components and examples. In *IEEE Symposium on VL and HCC*, pages 195--202, 2006.

[72] D.M. Syer, B. Adams, Y. Zou, and A.E. Hassan. Exploring the development of micro-apps: A case study on the blackberry and android platforms. In *SCAM'11*, pages 55--64, 2011.

[73] Mark Syer, Meiyappan Nagappan, Bram Adams, and Ahmed Hassan. Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In *CASCON 2013*, 2013.

[74] Ferdian Thung, David Lo, and Lingxiao Jiang. Detecting similar applications with collaborative tagging. In *ICSM'12*, 2012.

[75] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.

[76] Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03*, pages 408--418, 2003.

[77] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE '02*, pages 513--523, New York, NY, USA, 2002. ACM.