Summary:

Objective:

1. The goal of this paper to identify redundant features from the installed apps and understand the extent of feature redundancy on a smartphone. We first present a simple NLP-based method to identify common features from an Android app, and then evaluate the method on a set of popular apps. Once we successfully uncover the features within each app, we can study the extent of feature redundancy with a collection of used/installed apps from real smartphones.

Hypothesis:

1. • Features in our study must be widely used features
2. such as weather or app management, such that these
3. features are needed on most smartphones.
4. • These features may be included in apps that are not
5. designed for its main purposes. For example, although
6. many apps are not originally designed for weather,
7. they may include weather as an extra feature. Typical
8. apps fitting into this description include browsers, map
9. services, travel services, etc.
10. • The features may incur overhead or cost in resources
11. or battery if they are redundant. Take weather as an
12. example, multiple apps checking weather information
13. periodically will incur undesirable side-effects.

Limitation:

x

Detailed Summary:

Methodology:

* Software bloat is a process where successive versions of
* a computer program become perceptibly slower, use more
* memory, disk space or processing power than the previous
* versions [34]. It is originally a problem found in Windows
* PCs, and now become an increasingly important problem in
* mobile devices as well [27].
* Modern mobile systems are designed to prevent bloat,
* as it allows apps to share data with each other easily (for
* example using Intent [8] in Android). Mobile apps should be
* lightweight and streamlined for its own tasks, while ideally
* many small apps work together to complete complex tasks.
* However, the software bloat problem also exists in the
* mobile platforms. Due to security concerns or financial
* considerations, many mobile app developers are adding more
* and more features to their own app, instead of making use
* of the existing (same) features in other apps.
* App bloat is one of the main causes of feature redun
* C
* location: dynamic, static and textual analysis.
* Dynamic analysis examines the code traces that are
* actually executed at runtime, and then binds them with
* features observed. The code traces can be compared to each
* other in order to find feature-specific code [10, 35]. Static
* analysis examines the internal structure of programs like
* control or data flow dependencies. This is typically how
* feature locations are manually done: programmers jumps
* between files according to module dependencies [4]. Textual
* analysis infers the feature of code pieces by the naming
* information. Common machine learning and information
* retrieval techniques are used in textual analysis.
* V
* Approaches targeting traditional software mainly make
* use of text processing techniques such as LSI (Latent
* Semantic Indexing) and LDA (Latent Dirichlet Allocation)
* to determine the features in a software based on the textual
* information in code. For example, Mudablue [16] and Tian et
* al. [30] use LSI and LDA, respectively, to automatically
* categorize the open-source repositories, regarding a software
* system as a document and an identifier as a word. Similarly,
* Kuhn et al. [17], Baldi et al. [2] and Maskeri et al. [23]
* proposed text-based methods to extract topics from code.
* However, in modern software such as Android apps, textual
* information in code is often obfuscated in released versions.
* Xx different states based on the output events sensed by users.
* Another similar line of work is the detection of
* cloned/repackaged Android apps, such as DroidMOSS [37],
* DNADroid [7], or WuKong [32]. However, they are typically
* more coarse-grained than feature identification, as they
* mostly focus on detecting the similarity of whole apps, instead
* of components within them.
* Although there
* are a set of tools that can help kill the background tasks (task
* killers), the killed background processes may restart again
* and again by listening to some certain system events [11],
* thus even more resources might be consumed.
* Many approaches have been proposed to
* Android apps accurately.
* In this paper, we propose a keyword-based method to
* identify the features in Android apps. Our method is based
* on the insight that the user interface (UI) of an Android app
* often reflects the features within the app [18], particularly the
* UI text, which is closely related to the features. Moreover,
* unlike the identifier names in code which could be easily
* obfuscated, the UI text, which will be presented to end users,
* cannot be obfuscated. Thus our goal is to uncover the features
* of an app by matching the keywords related to each feature
* C
* n the app’s UI text.
* Our method can be summarized in three steps:
* (1) Extracting the UI text. We make use of existing reverse-
* engineering tools to decompile Android apps and
* extract the UI text. With the UI text, each app is
* converted to a text document.
* 3.2.2 UI Text Extraction. The UI of an Android app is
* mainly programmed with the XML format under the layout
* directory. The strings used in the layout XML, which are the
* content of views in the UI, are stored in the resource files of
* the APK, and also represented in the XML format.
* We make use of an existing reverse-engineering tool
* apktool 1 to obtain the resource files, and extract the key-
* value pairs from the res/values/string.xml file, which is the
* source of the app’s word set. The keys of the pairs are the
* definitions of each string, which represent the developers’
* perspective of the UI. The values will be presented in the
* actual UI, which are what users will see. Both the keys and values are useful text resources used in our approach.
* (2) Generating a keyword list for each specific feature. For
* each feature, we manually pick one or two keywords
* intuitively, and expand to more keywords based on the
* UI text extracted from a large set of apps in step 1.3.2.3 Keyword Generation. We use word2vec 2 in our
* keyword generation phase. Given some text data as a set of
* ordered word sequences, word2vec generates an N-dimension
* vector for every single word in the text data, where the
* vectors of words having similar meanings remain closer in
* the N-dimension space. Based on the vector set, we build a
* most-similar keyword set starting from one or two words.
* Similarly, we can generate a keyword set for each specific
* feature. First we extract the string key-value pairs from
* the decoded files (res/values/string.xml) of over 13,000 apk
* files from Google Play (which were crawled during our
* previous work [18]) as the set of ordered word sequences.
* Next, we perform some cleaning to the sequence set, such as
* breaking down the identifiers with camel case and underscore-
* connected naming to a word list, filtering out stop-words and
* words with lower tf-idf, stemming, etc. Then we put the
* sequence set as the training data into word2vec.
* Next, for each specific feature, we intuitively pick one or
* two words as the starting keyword set, then get a larger
* keyword set using word2vec.
* (3) Identifying the features within apps. With the list of
* keywords for each feature generated in step 2, we
* identify the features of an app by matching the keyword
* list of each feature. A feature is identified in the app if
* we can find enough related keywords in the app.
* 3.2.4 Feature Identification. To identify features in an app,
* we first extract a word set from the UI text, and perform
* the cleaning similar to the keyword generation phase. Then
* for each feature, we find the words appearing both in the
* feature’s keyword set and the app’s word set. A feature is
* found in an app only when the number of matched keywords
* is greater than a threshold, which is heuristically set as 3,
* based on our manual examination on some top apps.
* which requires more attention in our community.
* Dynamic features. During feature identification, we can
* only detect features residing in UIs written in static code.
* However, many mobile apps nowadays are loaded dynamically
* or relying on WebView or HTML5. For example, we did
* not find any of our features from a popular messaging app
* WeChat, because most of its features are dynamically loaded
* during runtime. Many other apps load their main features
* from web servers. In order to detect features from these apps,
* we need to incorporate dynamic analysis techniques in the
* process, using tools such as Monkey or DroidBot
* main features are not always possible.
* There are two types of redundancy: unnecessary features
* such as checking weather information, or scattered features
* such as app management and messaging services. Different
* redundant features may need different types of treatment. For
* example, unnecessary features such as weather information
* are mostly “true” redundancy and we typically only need to
* keep one weather forecast service on a smartphone.
* Redundant features such as app management are different.
* We may need to manage and update each and every app on
* the phone, but we do not want each app to check for their own
* updates or even updates for all apps. One possible solution
* to this issue is feature consolidation, where we have a central
* service to control the update of all apps on a smartphone.
* Although Google recently requires all apps installed from
* Google Play can only update themselves through Google
* Play, it does not require apps installed from other parties to
* do the same.

Experiment:

* Based on these criteria, we surveyed a list of top apps and
* selected the following six common feature:
* Table 1
* C
* ist of popular apps to evaluate the
* feature identification method. The apps are downloaded from
* two app stores: the official Google Play market and a popular
* third-party market CoolAPK 3 . We downloaded more than
* 2,000 apps from each market and use a list of around 4,000
* apps in our study after identical apps are removed based on
* their package names.
* The statistics of these apps are shown in Table 2. Note
* that during the evaluation process, we have used a different
* set of apps from the list of apps used to build the keyword
* list. Besides Google Play, we also included about 2,000 apps
* from a popular app market in China, because the users in
* our user study (Section 4) are mainly from China.
* We run the feature identification method on all 4,000 apps
* and get the list of features detected in each app. The results
* 4.1.1
* 50
* Set A
* Datasets. We use three sets of data in our study:
* Set B
* Set C
* Figure 5: The distribution of the number of apps in
* each dataset.
* • Set A. One set of data is from a popular Android app
* market 4 , which includes the list of apps used by each
* smartphone user during the period of one week. The
* data includes used app lists from 600,000 users.
* • Set B. Heavy users from Set A. Because many users
* may not use many apps within one particular week,
* while users typically use fewer apps than the installed
* apps on their phones, we choose the top 10% of the
* users in the first dataset as the heavy users and study
* the extent of feature redundancy on these users. The
* dataset consists of 60,000 users and we expect that
* they can represent the typical heavy users with higher
* redundancy than Set A.
* • Set C. Another set of data is collected with a simple
* Android app retrieving the list of apps installed
* on a smartphone, which was designed by ourselves.
* The users we recruited are mostly volunteers from
* universities. Most of them are college or graduate
* students, such that they know basic information on app
* features and are thus capable of answering a simple
* survey on whether these features are necessary for
* each app. We have recruited 87 volunteers who have
* returned valid app lists in this dataset.
* Redundancy Calculation: to examine whether there are redundant features on a
* smartphone, we compare the list of apps on each smartphone
* with the list of 4,000 apps we have already studied in the
* last section. For each user, we record the features identified
* in all the apps matching the list of apps in our repository.
* We then count for each feature, how many of them can be
* found on every smartphone. If more than two apps
* users in each dataset.
* After we identified the features on mobile apps, we then
* perform a simple user survey, asking each user to indicate
* which features they think are unnecessary for each app. We
* ask each user to provide their opinion on at most 10 apps.
* of the survey.
* The survey results are shown in Table 8, which shows the
* number of features listed in the questions and the number
* of features indicated as unnecessary. Overall, out of 1,336
* features surveyed, users selected about 45% of them as
* unnecessary. For features such as “schedule”, about 65%
* of the occurrences are deemed as unnecessary by users, while
* for “map/location”, only 16% are considered as unnecessary.

Results:

* hin an app, and
* evaluated it on over four thousand popular apps. Experiments
* on a list of apps installed on actual smartphones show that the
* extent of feature redundancy is very high. We found that more
* than 85% of user smartphones contain redundant features,
* ded from both Google Play
* and a popular third-party market. Out of the 4,000 apps,
* we have identified more than 3,200 instances of the six
* popular features we studied. About 22% of the studied
* apps contain more than one potentially redundant features.
* Manual examination on a list of 50 top apps shows that the
* algorithm achieves a precision of almost 90%.
* ask users to indicate
* which features are unnecessary for each app. The survey result
* shows that 45% of the features surveyed are not desired from
* end users’ perspective. The most undesirable features are
* schedule, app management, and email/messaging services.
* • We propose a method to identify common features
* within an Android app. The method has been evaluated
* on more than 4,000 popular apps, with an identification
* accuracy of about 90%.
* are shown in Figure 2.
* Overall, we have detected that over 22% of apps contain
* the “schedule” feature, while almost 17% of apps contain
* the “app management” feature. “Weather” is also a popular
* feature in about 12% of the apps, while “news” only resides
* in about 5% of all the apps. In total, we detected more than
* 3,200 features, averaging about 0.8 features for each app.
* C
* Accuracy: n reading app description, running
* the app manually and checking the reasons why it is detected
* (whether the keywords are used for other purposes).
* The evaluation results are shown in Figure 3. We show for
* each app all the features detected and whether it is correct or
* incorrect. We also summarize the overall detection accuracy
* data in Table 3. For all the 94 features identified from these
* 50 apps, 89% of them are correctly identified. Our detection
* algorithm is perfect on “schedule” and “news”, while only
* have a 82% precision on detecting the “weather” feature.
* Feature Overlapping. We then examine how features
* overlap with each other in Table 4. For all apps detected with
* each feature (in each column), we show how many of other
* feature (in each row) can be detected in these apps.
* Overall, the overlapping rate ranges from 10% to 49% for
* any pair of the six features. Take weather as an example, for
* all 475 apps containing the weather feature, 215 (45.3%) of
* them contain “schedule”, while 140 (29.5%) of them contain
* “app management”.
* C
* 4.1.2 Dataset Comparison. Figure 5 shows the difference
* in three datasets. In Set A, we can see that about half of
* the users have used fewer than 50 apps, while 90% of the
* users have used fewer than 90 apps during the period. In
* comparison, 90% of all users from Set C have installed more
* than 150 apps (many of them are pre-installed apps by the
* phone manufacturer), while 10% of the users have installed
* more than 350 apps!
* Although the numb
* For all datasets, we can see that “app management” is
* the most redundant feature, averaging between 5 to 12 in
* the three datasets. The reason is because that Android apps
* installed from third-party markets or even the app’s own
* websites typically check updates themselves. For one extreme
* case, we found over 100 apps containing app management
* features on a single smartphone.
* close to the level of redundancy in Set C.
* Table 7 shows the number of users with different numbers
* of redundant features from the three datasets. More than 85%
* of users from Set A contain at least one redundant feature on
* their phones, while all but one user (more than 98%) from
* Set C contain at least one redundant feature. Almost all users
* from Set B contain feature redundancy: it is not surprising
* because the users are top users selected from Set A.

Positive Points:

1. Released Source Code.
2. OS level Solution (maintaining versatility)
3. Verified implementation lacking among different naive solution and approaches not just experiment but at the level of comparing source code.

Negative Points:

1. Did not explain the energy consumption and battery overhead.
2. Did not provide the actual strength of the survey conducted.
3. Did not explain interface interaction and switching at the implementation level.
4. Limited features and apps.