Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications

Steve Hanna[1], Ling Huang[2], Edward Wu[1], Saung Li[1], Charles Chen[1], and Dawn Song[1]

[1] UC Berkeley
[2] Intel Labs

Abstract. Mobile application markets such as the Android Marketplace and the Amazon Android store provide a centralized showcase of applications that end users can purchase or download for free onto their mobile phones. Despite the influx of applications to the markets, applications are either largely unreviewed or only cursorily reviewed by marketplace maintainers due to the vast number of submissions; furthermore, they rely on user policing and reporting to detect misbehaving applications. This reactive approach to application security, especially when programs can contain bugs, malware, or pirated (inauthentic) code, puts too much responsibility on the end users.

In light of this, we propose Juxtapp, a scalable infrastructure for code similarity analysis among Android applications. Juxtapp provides a key solution to a number of problems in Android security, including determining if apps contain copies of buggy code, have significant code reuse that indicates piracy, or are instances of known malware. We evaluate our system using more than 58,000 Android applications and demonstrate that our system scales well and is effective. Our results show that Juxtapp is able to detect: 1) *463* applications with confirmed buggy code reuse of Google-provided sample code that lead to serious vulnerabilities in real-world apps, 2) *34* instances of known malware and variants (including *13* distinct variants of the GoldDream malware), and 3) pirated variants of a popular paid game.

1 Introduction

As mobile devices (e.g., smartphones, tablets) gain popularity, software marketplaces have become centralized locations for users to download applications. For the Android operating system, Google hosts the official Android Market while Amazon and many others provide third party markets. The wide range of devices that are Android-compatible combined with the open source nature of the Android operating system and development platform have led to explosive growth of the Android market share. As of August of 2011, Android has grown to a 52% market share[1].

The rapidly increasing volume of applications, increased demand for diversified functionality, and existence of piracy and malware places large obstacles in the way of a healthy and sustainable Android market.

Vulnerable code reuse. Android developers often misuse coding idioms in Android, either due to copying and pasting of vulnerable code or lack of developer understanding[2, 3]. For instance, Google has provided sample code to interface with the License

Verification Library and the In-Application Billing APIs, which are responsible for verifying that a user is authorized to execute a program and purchasing virtual items within an application, respectively[4, 5]. Google explicitly warns developers that they need to modify certain parts of the code, because the unmodified template code is subject to certain security vulnerabilities and requires developer intervention in order to ensure security properties.

Malware. With the exploding growth in the number of Android applications, the occurrence of Android malware has also increased. As of August 2011, users are 2.5 times more likely to encounter malware on their mobile devices than only 6 months ago and it is estimated that as high as 1 million users have been exposed to malware[6].

Piracy. Furthermore, the Android software marketplaces are home to many pirated applications. A common occurrence is for an illegitimate author to repackage and rebrand a paid or popular app with additional program functionality in order to generate revenue and even execute malicious code[7].

The current markets usually rely on two approaches to identify and remove potentially dangerous applications: 1) review-based approach, which requires mostly expert manual review and security examination, and 2) reactive approach, e.g., user policing, reporting, and user ratings as indicators that an application may be misleading in its functionality or misbehaving. Given the existence of hundreds of thousands of applications on the markets, neither approach is scalable and reliable enough to mitigate threats to users. To empower and expedite this process, we need an automated analysis of Android applications in order to pare down large application datasets into a small set of noteworthy candidates for further investigation.

Each of the aforementioned problems appears to be unrelated. However, we observe a common invariant among them, namely, code reuse, which sheds light on the fact that a unified approach in detecting common code (or code similarity) may address all of our goals. Using this observation, we propose to build a fast and scalable infrastructure for detecting code reuse in Android applications which allows for 1) early detection and developer notification of known vulnerable or buggy code, 2) detection of instances of known malware, either in isolation or repackaged with an innocuous program, and 3) detection of pirated applications.

It is a challenging task to develop a system to automatically detect code reuse in Android applications. The system must be able to quickly compare code and detect reuse, and scale to hundreds of thousands applications or more; the system need to be resilient to certain levels of code modification and obfuscation, which are common in Android applications; the system should be able to represent the application being compared in a meaningful, accurate way in order to find the so-called needle-in-a-haystack differences in applications, all the while maintaining low false positive and false negative rates.

As a first step solution, we use k-grams of opcode sequences of compiled applications and feature hashing[8, 9] to efficiently tackle the problem at large-scale. k-grams of opcode sequences have been shown to be resilient to certain types of code modification and can be efficiently extracted from applications. Additionally, feature hashing has been shown to work well in dimensionality reduction and classification. We combine this technique with a variety of domain-specific knowledge in order evaluate code reuse, instances of known malware, and piracy in Android applications. We use k-grams and

feature hashing combined in order to have a robust and efficient representation of applications. Using this representation, we have a fast way to compute pairwise similarity between applications to detect code reuse among hundreds of thousand of applications.

However, a simple question remains: *What about using signature matching?*. Using signature matching tools, like grep or other more sophisticated techniques can be a way to determine code reuse or known malware in a dataset; however, the analyst is responsible for picking the features to search for, which presupposes that the most important features are known a priori, which is simply not the case. Furthermore, code mutations, string modifications, class name changes, etc. would all be missed by grep except in the simplest of cases.

In this paper, we propose Juxtapp, a scalable architecture for quickly detecting code reuse and similarity in Android applications. We implemented our distributed architecture using Hadoop and ran it on Amazon EC2. It is capable of fast, *incremental* additions to the analysis dataset, meaning it is amenable to frequent updates and additions to the pool of applications. We apply Juxtapp to address three different types of problems: vulnerable code reuse, known malware, piracy. We evaluate Juxtapp's ability to detect these problems on 58,000 applications, ranging in size from hundreds of kilobytes to tens of megabytes, which were collected from the official Android market and the Anzhi third party market[10]. We find that the system performs and scales well.

- *Vulnerable Code Reuse*. We show that applications widely use significant portions of the Google In-App Billing and License Verification example code, leaving them susceptible to vulnerabilities.
- *Instances of Known Malware*. We find **34** instances of malware in Android markets, **13** of which are distinct, previously unknown variants that have been repackaged with innocuous-looking applications.
- *Piracy*. We identify pirated applications in third party markets and show that Juxtapp can detect pirated applications that are obfuscated and with significant code variation from the original application.

2 Problem Definition

In this paper we consider the problem of automatically finding similarity among Android applications with the goal of detecting known buggy code patterns and vulnerabilities, repackaged and pirated applications, and known malware in Android markets. Detecting code reuse in Android applications offers a first chance in detecting applications that may negatively impact the user's security and experience or defraud developers of revenue. We develop Juxtapp, an architecture that automatically examines code containment in Android applications. We define code containment to be a measure of the relative amount of code in common between two Android applications. Using this, we examine a variety of Android market applications for instances of vulnerable code, known malware, and piracy.

Buggy and Vulnerable Code Reuse. Previous manual investigations into developer errors[3, 2] in Android applications have indicated that developers often copy and paste code as well as reuse sample code obtained from Android-specific developer websites

without modification. Using application similarity, we can examine the Android Market to see if they contain known buggy or vulnerable pieces of code.

Known Malware. The incidence of malware in Android marketplaces has been rising rapidly. In January 2011, 80 applications were known to be infected with malware, as opposed to June 2011, when the incidence rate had risen to over 400 instances of malicious applications [6]. Malware authors often repackage legitimate applications with a malicious payload in order to entice users to download an infected application.

Piracy and Application Repackaging. Popular Android applications and games are commonly repackaged with modified code in order to evade copyrights protection and to generate revenue for the pirate [7]. By comparing applications from the official Android market to third party markets we show that we can detect instances of piracy.

Scope. We restrict ourselves to the Android application domain, excluding obfuscation in the form of functional code transformation. For instance, we are able to detect two instances of similar obfuscated code, but we restrict ourselves to this domain and do not consider the problem of matching code which has been transformed to be functionally equivalent.

2.1 Goals and Challenges

Juxtapp has a variety of challenges which must be met in order to detect code reuse in Android applications. Some specific goals of our platform are to:

Automatically analyze code similarity in Android applications. As of November 2011, the Android market had over 310,000 applications[11]. From mid-May 2011 to November 2011, the Android Market gained 100,000 applications, increasing the the number of available applications by 50%. This rapid growth of market applications and increase in the number of pirated and malicious applications underlines the need for a way to rapidly and automatically analyze applications.

Scale to a large number of applications. As previously stated, Android markets have hundreds of thousands of applications with new applications being added all the time. Our architecture must be able to scale with market growth in order to detect similarity across a wide range of applications. This includes being able to *incrementally* update our application respository in an efficient manner.

Accurately and efficiently represent the applications under analysis. In handling hundreds of thousands of applications, Juxtapp must be able to accurately represent and quickly determine code similarity among applications. There is an implicit trade-off between the accuracy of the analysis and the amount of space it takes to represent an application under analysis.

Android Specific In addition to general challenges, there are a number of domain specific considerations when computing the similarity of Android applications.

Java Source Code Unavailable. For most applications on the Android Markets, source code is not available. Android applications are compiled from Java to Dalvik bytecode

(known as the DEX format), which is the bytecode for the Dalvik virtual machine[12]. This compiled code and application resources are packaged as an application package, known as an APK.

Multiple Entry Points. Unlike traditional desktop programs, Android applications have multiple potential entry points. Android applications are broken up into components and these components can each have their own entry points, invokable by the Android inter-process communication mechanism known as Intents. Therefore, any type of program analysis must take this into consideration.

Repeated Inclusion of External Libraries. Android applications are in an application sandbox within the Android operating system. This means that each application must include the libraries required to operate the application. This leads to significant code copying across applications. For instance, advertising libraries such as AdMob, which often comprise of a significant fraction of the total code within an application, are repeatedly included across applications. Juxtapp must consider this fact when determining similarity because these types of libraries can dominate the analysis of application similarity.

All resources required to run application in APK. Inside the archive are all of the dependencies required to install and use the application. Android applications are written in Java and then compiled to DEX, the Dalvik virtual machine executable format. All of the Java application code is contained in the `classes.dex` file, which we extract, process and feature hash. We discuss the details of this step in Section 4.1.

Obfuscation. Android developers are encouraged to obfuscate their code using Proguard[13]. Proguard attempts to remove unused code and renames classes, methods, and fields with obscured names to make reverse engineering of Android applications more difficult. However, this process is deterministic so two identical applications will be transformed in the same way. As previously stated, we consider obfuscated code, and can match similarly obfuscated code, but we restrict ourselves to not considering matching functionally equivalent code snippets.

3 Background

Like static code reuse detection proposed in [14, 15], we use k-gram features of code sequence to represent applications. However, k-grams extracted from code sequence usually results in an enormous feature space (e.g., 2^{128} features in [14, 15]), preventing efficient feature storage and similarity comparison even for a moderate number of applications. To analyze large volumes of mobile applications, we need an efficient feature representation of the applications and a fast way to compare features between them.

Feature Hashing. The main technique we use is feature hashing. Feature hashing is a popular and powerful technique for reducing the dimensionality of the data being analyzed [8, 16]. Using a single hash function, it compresses the original large data space into a smaller, randomized feature space, in which feature hashing, representation, and pairwise comparison are all efficient. This efficiency comes at the cost of potential collisions while hashing. However, theoretical and experimental results from the machine

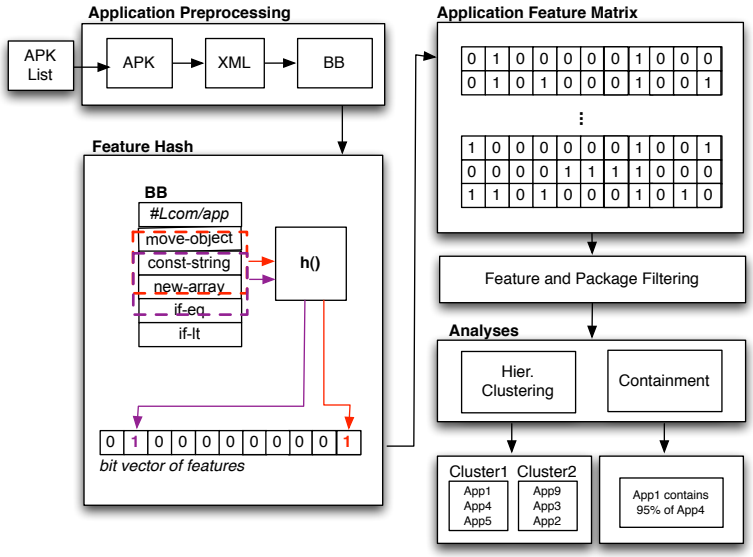

Fig. 1. The Juxtapp Workflow

learning community show that pairwise similarity maintains high accuracy, thus algorithms built on top like hierarchical clustering, will be close to exact [8, 16]. Feature hashing was recently introduced into the security community for malware analysis [9].

The resulting representation of an application can be encoded into a succinct bitvector which represents the features present in the data. As always, choosing a good hash function and a bitvector representation of prime length is essential to minimize the number of collisions in the vector.

Similarity. We determine the similarity of two applications by the similarity between their feature sets. We use the Jaccard similarity metric defined as $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$, where A and B are two k-gram feature sets of two applications, respectively. Because we hash k-gram sets into boolean vectors, with each entry indicating presence or absence of a feature, as opposed to a set of items, we can approximate this quantity much more efficiently using bit-wise operations: $J(\hat{A}, \hat{B}) = \frac{|\hat{A} \wedge \hat{B}|}{|\hat{A} \vee \hat{B}|}$, where $\hat{A}$ and $\hat{B}$ are bitvector representations of k-gram sets A and B, respectively. As shown in [9], as long as the size of the bitvector is large enough, $J(\hat{A}, \hat{B})$ is very close to J(A,B), the similarity between two applications represented by the k-gram feature sets. The Jaccard distance $D(A, B)$, which measures *dissimilarity* between two feature vectors, is obtained by subtracting the Jaccard similarity from one: $D(\hat{A}, \hat{B}) = 1 - J(\hat{A}, \hat{B})$. Both Jaccard similarity and distance have values in the range $[0, 1]$.

4 Our Approach

As shown in Figure 1, our approach, Juxtapp, involves the following steps for analyzing Android applications: 1) application preprocessing, 2) feature extraction, and 3) clustering and containment analyses.

4.1 Application Preprocessing

We preprocess each application in order to extract the DEX file, which represents the compiled application code. In our approach, the original Java source code is *not required* because the DEX format fully describes the application and retains class structure, function information, etc.

For each application we convert its DEX file into a complete XML representation of the Dalvik program, including program structure. From this, we extract each basic block and label it according to which package it came from within the application. We process each basic block and only retain the opcodes while discarding most operands. The exception to this is for opcodes storing constant data, such as the `const-string` opcode, which becomes a concatenation of the opcode along with the value it references.

The intuition behind this is that many Java applications contain boiler plate code that will appear in many applications when only opcodes are considered. Furthermore, including constants makes the feature hashing (discussed below) more fine-grained and more restrictive about matching. This is especially important because many applications use Java reflections to access functionality, with the only difference between programs being the string arguments passed to the Reflections API.

Additionally, Juxtapp is optionally able to exclude packages and features from the analysis in order to reduce false matched of very common code. The details of this step are discussed in Section 4.4.

4.2 Feature Extraction

We use k-grams of opcodes and feature hashing to extract features from applications. We use the `djb2` hash function which is known to have an excellent distribution[17]. As shown in the *Feature Hash* box in Figure 1, for each application's basic block representation of the original XML file, file), we extract each k-gram using a moving window of size k, and hash it using `djb2`. k-grams across basic blocks are ignored. For each hashed value, we set the corresponding bit in the bitvector of the application, indicating existence of the k-gram. Along with this information, we efficiently store the package name from which the basic block originated, the basic block offset within the basic block file, and the k-gram offset. This allows us to recover how and why our architecture determined that applications are similar and serves as a way to verify matching applications.

In order to feature hash, we have two parameters to determine, namely: length of k-gram k and bitvector size m. In Section 5, we show an experimental evaluation of several values of k and m in order to determine optimal values of these parameters over our dataset.

Choosing k. k is a parameter which determines the number of dimensions of the underlying feature space for representing the Android applications, and it bounds the number of features that can be extracted for each application. k is a crucial parameter for detecting similarity. If k is too small (e.g., $k = 1$), there will be a small number of unique features from all applications, resulting in an oversimplified, low-dimensional representation of the applications. In this representation, overmatching between applications can occur, and many applications would be falsely classified as being similar applications would be falsely similar. On the other hand, if k is too large (e.g., bigger than the size of most basic blocks), we would obtain a very high-dimensional representation of the applications. However, because we may only be able to extract a few features from each application due to the high dimensionality of the data, using a large k limits the ability to have meaningful and robust comparisons between applications. In general, a reasonable k should have a small value, at which further increase in value would cause insignificant increase in the quality of the pairwise similarity comparison. As shown in Section 5, we evaluate a set of different k values, and choose $k = 5$, at which its marginal impact on similarity accuracy is around 0.01.

Choosing Bitvector Size. The bitvector size m strikes the tradeoff between (similarity) computation efficiency and approximation error of the bitvector representation of the k-gram features.

Ideally, we want size m to be large enough so that few collisions would happen when we feature hash k-grams into bitvectors; practically, we want size m to be small so that we can efficiently compute pairwise similarity among hundreds of thousands of applications. The larger the bitvector size m, the more accurately a bitvector represents an application, but at the cost of more time required to compute the pairwise similarity among all applications.

As shown in [9], as long as $m \gg N$, which is the number of k-grams extracted from an application, the Jaccard similarity between two bitvectors very closely approximates computing the set intersection between two k-gram feature sets. That is, as long as m is large enough, Jaccard similarity is nearly an exact representation in practice. Based on this principle, we use a data-driven approach in our experiments in Section 5, in order to determine a bitvector size which is large enough to represent the feature space in question. Large enough means that pairwise Jaccard similarity is accurate, yet still allows for efficient computation of similarity between all pairs of hundreds of thousands of applications.

4.3 Analysis of Feature Hashing Results

A variety of data analyses can be performed on the feature representation of the applications. In this paper, we primarily focus on similarity, containment and clustering analysis, which help us to filter out vast amounts of uninteresting instances and pare down a small set of interesting candidates for further analysis.

Code Containment Comparison Containment analysis is a useful tool for paring down application candidates that potentially have copied code, pirating, and malware contamination. We define the containee A to be the application being examined for

similarity and the container (or carrier [3]) B to be the application which houses the packages and associated features that we test for existence inside the containee. We define a metric that gives the percentage of containment by considering the number of features common in both applications, divided by the number of bits in the containee application. Formally, containment is defined as: $C(\hat{A}|\hat{B}) = \frac{|\hat{A} \wedge \hat{B}|}{|\hat{A}|}$. Written in this form, this containment is defined as the percentage of features in application A that exist within application B.

Clustering To find inherent patterns among Android applications, we use agglomerative hierarchical clustering[18] on the feature bitvector representation of each application in order to group similar applications together. The basic idea is that the collection of feature bitvectors represents the applications in a high-dimensional space with a well-defined distance metric, the Jaccard distance. Using this distance metric, we can group bitvectors that are close-by and, thus, we are able to group similar applications.

Hierarchical clustering produces clusters without having to specify the number of clusters in advance. The input to the clustering algorithm is a threshold t (e.g., 90%) and a list of Jaccard similarity values between each pair of applications. The output is a clustering S for the applications, in which all applications in a cluster are with similarity s greater than or equal to $t : s \geq t$. The threshold t is set by the desired precision tradeoff between the number of applications in the clusters and the "closeness" of applications within a given cluster. While a smaller t puts more applications into a few large clusters, a larger t discovers specific variants of application families (e.g., similar applications developed by the same authors).

Hierarchical clustering begins with one application in its own cluster; then it selects the closest pair and merges them into a common cluster. The cluster comparing and merging process continues until there is no pair whose similarity exceeds the input threshold t. For clusters with multiple applications, we use single-linkage to define the similarity between them, i.e., the similarity between cluster S_a and cluster S_b is the maximum similarity between all possible pairs, i.e., $J(S_a, S_b) = max\{J(A, B)|A \in S_a, B \in S_b\}$.

4.4 Core Functionality and Result Refinement

Clustering can be a way to visualize the application topology in order to qualitatively understand how well applications are classified among a given cluster. Application similarity can be dominated by large similar libraries common to many applications (i.e. AdMob). In light of this, we develop the notion of *core functionality*, which seeks to capture in a coarse-grained manner how included libraries interact with the main application component.

Simply put, we examine each application and whether or not the core application component directly invokes an outside library. If it does then it is a part of the application's functionality; otherwise, that code can be excluded from our analysis. We

[3] In the case of malware, a carrier is a more appropriate term because the innocuous application is modified in order to execute code outside of the intended functionality.

refer to the set of libraries excluded as an *exclusion list*. We point out that this is an over-approximation and aggressively excludes libraries due to Java reflection as well as dynamically registered event handlers, and other entry points defined by the Android Manifest. All of these are additional entry points that can be invoked by external entities. For the purpose of our analysis, we restrict the entry point to the main application entry point defined in each application Manifest. We consider improving this in future work; however, this methodology is already helpful in visually capturing and characterizing the quality of our clustering.

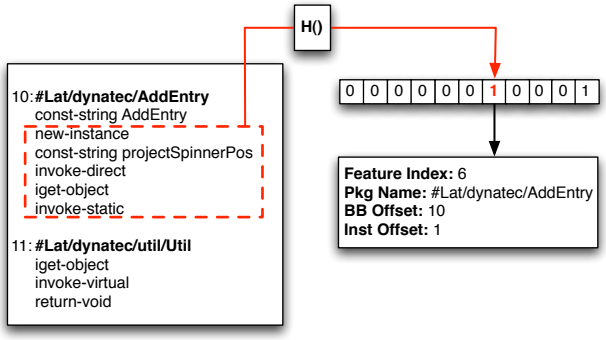

Fig. 2. Example outcome of feature hashing a basic block found in an application.

4.5 Implementation

The workflow of Juxtapp can be roughly broken up into the following stages: application preprocessing, feature hashing, clustering and containment analysis. Juxtapp consists of 6,400 lines of C++, 1,600 lines of Java, and 600 lines of scripts.

The first step in the process is converting the Android application file (APK) to a format which is usable by our architecture. The APK file contains the `classes.dex` file which contains the Dalvik representation of the Java code of the application. Juxtapp processes this file and outputs the file in an XML format with functions split into basic blocks. From the XML file, we convert it into the basic block format, which outputs the opcodes contained within a basic block along with a label indicating the source package, class and method.

Next, after preprocessing, the applications are feature hashed. Juxtapp processes the basic block file for each application and outputs a feature vector representing the application along with recovery information to verify matching portions of applications That is, in addition to the features, we also store the package and class name, and the offset within the original file in order to verify matching potions of applications. Figure 2 shows an example basic block being feature hashed, along with the recovery information we store for each feature, which is required by further analysis. For each program under analysis, the features calculated are stored as a sparse representation of the vector

in a `fh file`, while a table of each feature's offset within the original program and the package and class from which it originated are stored in a `tbl file`. The feature hash file stores the size of the vector, the number of entries that are set to one, and a list of features appearing in the program. The table file stores a feature number, the basic block offset within the file, the instruction offset within that basic block, and finally the class, package, and method name that the basic block was derived from.

After processing all of the applications' basic block files, Juxtapp calculates a pairwise distance matrix between all applications. This matrix is used for clustering and determining similarity among applications. We note that Juxtapp is capable of incremental analysis which allows additional applications to be considered without recalculating the entire dataset.

After the applications have been feature hashed, Juxtapp can perform other in-depth analysis. First, the applications under analysis can be clustered based on their computed distance matrix, which offers a topological view of the dataset, which can help an analyst narrow down interesting areas to investigate. For instance, after clustering, we combine author information with the applications in each cluster. This allows us to understand which applications came from which authors and helps us identify interesting candidates, i.e., those with conflicting authors and similar but modified application code.

Finally, Juxtapp computes containment between sets of applications. Given a set of feature hashed applications represented by their `fh` and `tbl` files, the containment tool determines what features are common between applications and outputs the percentage of code in common, along with the ratio of the comparative sizes of the number of features. The intuition behind this is that a large application when compared to a small application may inadvertently have a large subset of the smaller applications features by virtue of the fact that a larger application will produce a dense feature vector. This ratio is used to remove false positives due to bit vector collisions.

Distributed Analysis. We have both a single machine implementation of Juxtapp as well as a distributed implementation which uses Hadoop on Amazon EC2. We use the Hadoop MapReduce framework for performing large-scale computations and HDFS for sharing common data among nodes[19]. We wrote a MapReduce application in order to perform the APK to Basic-Block conversion portion of the workflow, and we used Hadoop Streaming to interface with our C++ applications, which were responsible for feature hashing and containment calculations. We note that the Hadoop Streaming interface is unable to take advantage of resource management because of the externalities of the program being interfaced[4]. However, Many of the tasks required of Juxtapp are easily parellizable tasks which greatly improves performance when dealing with large datasets. As a result, Juxtapp can feature hash, cluster, and analyze containment in a distributed manner which offers great performance increases over the single machine version.

Incremental Update. The statelessness property of many stages in Juxtapp makes it easy to incrementally process the applications, update their similarity matrix, and analyze them in detail without the need to reprocess all applications under analysis. When

[4] Hadoop Streaming uses program input and output to interface with the Java application.

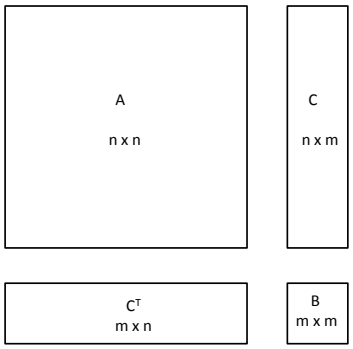

Fig. 3. Incrementally updating the similarity matrix. A contains similarity values among existing applications, B contains similarity values among the new applications, and C contains similarity values between the new applications and the existing ones.

a set of m new applications are added to the analysis, the application preprocessing (the conversion of APK files to XML file to Basic Block files) and the feature hashing are inherently incremental, meaning, only the new applications need conversion and feature hashing. As shown in Figure 3, with n existing applications and m new applications, updating the existing $n \times n$ similarity matrix A is straight forward as follows: 1) compute the $m \times m$ similarity matrix B among the new applications, 2) compute the $n \times m$ similarity matrix C between the set of new applications and the existing ones, and 3) concatenate them together and grow the existing similarity matrix A at appropriate rows and columns to get the new $(n + m) \times (n + m)$ similarity matrix.

With the new similarity in B and C, it is also straightforward to update the hierarchical clustering using incremental methods to obtain a new clustering results [20, 21].

5 Evaluation

In this section, we evaluate the efficacy of Juxtapp. We first introduce our evaluation dataset and describe our experimental setup. Then, we discuss determining experimental parameters and their impact on our results. Finally, we introduce case studies in which we use Juxtappto detect instances of vulnerable code reuse, known malware, and piracy on Android markets.

5.1 Experimental Evaluation Dataset

We evaluate our approach using applications from three different sources. From the official Android Market we obtained 30,000 free Android applications. Additionally, we downloaded 28,159 applications from a third-party Chinese market, Anzhi [10], and the 72 malware in our malware dataset came from the Contagio malware dump and

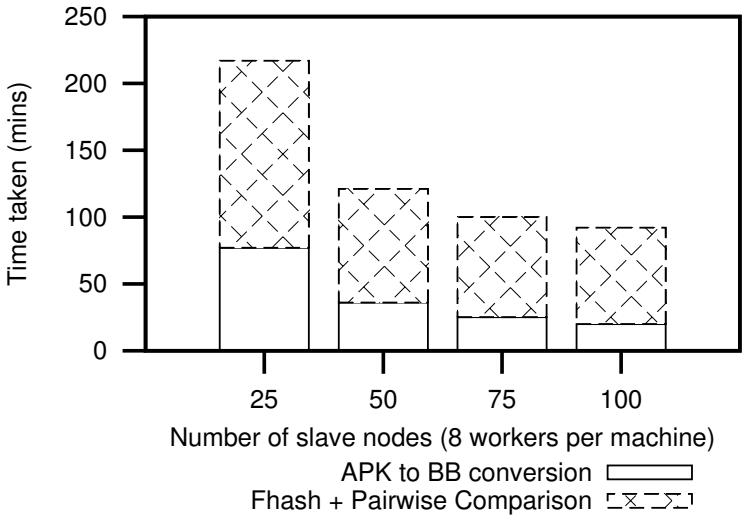

Fig. 4. The running time of our complete pipeline with various number of workers per cluster on Amazon EC2. Time are measured when processing 95,000 unique Android applications.

other sources [22]. Lastly, we use a set of 95,000 Android applications from the official Android Market to evaluate the performance of Juxtapp [5].

5.2 Experimental Setup and Performance

Local experiments, when tractable, such as containment between a small set of applications and our dataset, were run on Ubuntu Linux 2.6.38 with Intel Xeon CPU (8 cores) and 8GB of RAM. When larger experiments were required, such as containment between on-market to off-market applications, and generating pairwise distance matrix, we conducted them on Amazon EC2. For our Amazon EC2 clusters, we used m2.4xlarge instances, which run on Ubuntu Linux 2.6.38-8-virtual with 8 virtual cores and 68.4GB of memory.

We varied the number of nodes running from 25 to 100 and used 8 worker threads per node. Figure 4 shows the time required to complete a full run of the entire pipeline, which includes APK to basic block format file conversion, feature hashing, and computation of the pairwise similarity when using 95,000 unique Android applications. At the time of writing, there are around 310,000 Android Applications[11], which demonstrates that Juxtapp scales well.

As we increase the number of nodes, the amount of time required to do analysis becomes gradually dominated by the overhead of parallelization. In addition, the APK to Basic-Block and feature hashing stages were parallelizable without any synchronized state, which contributed to significant performance gains as the number of

[5] We obtained a larger dataset of applications in order to show that our technique scales to a large number of applications beyond our evaluation set of applications

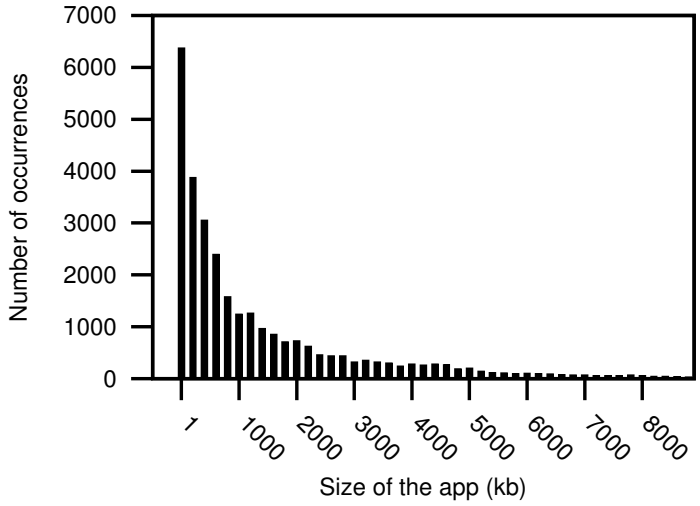

Fig. 5. Frequency distribution of size of APK files. For better visualization, we do not show the largest 4% of the applications. The largest APK file in our dataset is 52.26MB. The numbers on the x-axis are the lower bounds of the bins, and the size of each bin is 200.

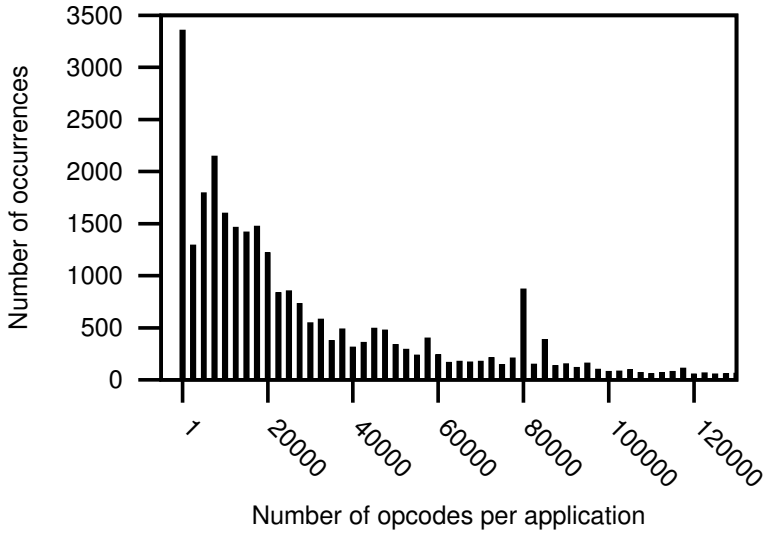

Fig. 6. Frequency distribution of number of opcodes of applications. For better visualization, we do not show the largest 11.2% of basic block files. The largest file in our dataset has 1,728,196 opcodes. The numbers on the x-axis are the lower bounds of the bins, and the size of each bin is 2500.

workers increased. However, the pairwise distance comparison is the current bottleneck on performance because it combines the resulting bitvectors from each worker. Figure

k	Avg. Dist
3	0.939
5	0.969
7	0.980
9	0.984

Fig. 7. Experiment showing the impact of varying k on the Jaccard distance.

4 shows how the overhead of the pairwise compairison approaches a constant overhead as the number of nodes are increased.

Incremental Update Performance. Incremental updates of the dataset allow us to continuously process and update our dataset with new market applications without requiring running the entire Juxtapp workflow on our application repository. Table 1 shows the time required to add from 100 to 7,000 APKs to the dataset. Distribution time is the time required to distribute APKs to worker nodes. This time begins to become dominant as the number of APKs increases. This overhead is caused by not being fully able to take advantage of Hadoop's resource allocation, due to our Hadoop Streaming implementation. Despite this, these numbers show that adding a large number of applications to the comparison repository daily or even multiple times daily is feasible with Juxtapp.

# Incr. APKs	Distribution Time	Completion Time
100	0m 36s	5m 11s
500	4m 49s	9m 35s
1000	8m 58s	21m 5s
3000	20m 20s	42m 31s
5000	42m 52s	80m 51s
7000	57m 0s	104m 48s

Table 1. The time to incrementally process varying numbers of APKs. Note, distribution time is included to show how file distribution starts to dominate the processing time.

5.3 Dataset Statistics

To gain a general understanding of our dataset, we analyzed our collection of 30,000 unique applications as a representative sample of the official Android Market. Figure 5 shows the distribution of the sizes of APK files in kilobytes, and Figure 6 shows the distribution of the number of opcodes per application. Both distributions are skewed to the right, with APK files having a median size of 724KB and applications having a median number of opcodes of 20,555. The 75th percentile values for APK file sizes and number of opcodes are 2,071kb and 56,166, respectively. The total file size of these APKs is 50.43GB and total number of opcodes in all applications is approximately 1.45 billion.

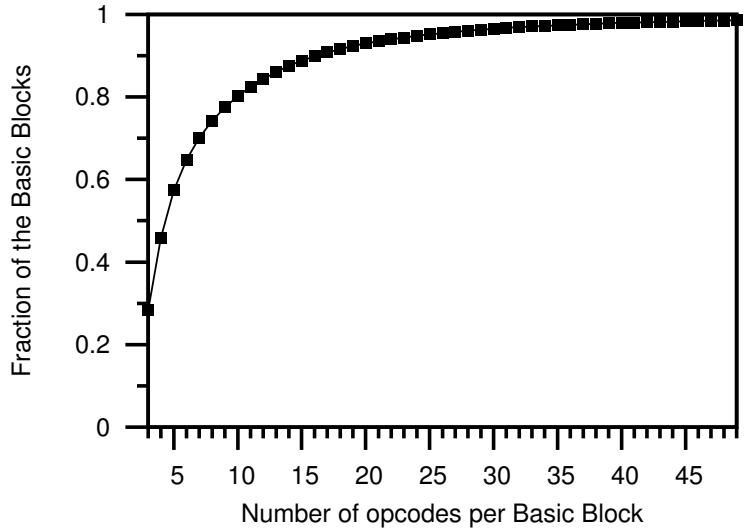

Fig. 8. Cumulative distribution of the number of opcodes per basic block, using all basic blocks with more than 2 opcodes. For better visualization, we do not show the largest 1.5% of the basic blocks. The mean is 5.35 opcodes and the median is 2 opcodes, while the largest Basic Block in our dataset contains 35,517 opcodes.

5.4 Determining Experimental Values

Before feature hashing we must choose values for k-gram size k and bitvector size m. We use the 30,000 Android applications to determine their values.

Choosing k for our Dataset. To choose k, we randomly select pairs of applications and evaluate their Jaccard distance to determine how much varying k impacts the average distance between them. Figure 7 shows varying values of k and the resulting average distance between pairs of randomly sampled 6,000 applications[6]. We repeat the experiment on multiple runs, but see little variance across them. The key intuition is if two applications are chosen at random from our dataset, they are likely to be dissimilar. The table shows that starting from 5, further increasing k has little impact on the distance calculation. Based on this, we chose a value of k to be 5 and performed feature hashing and clustering on our sampled applications. Figure 8 shows the cumulative distribution of opcodes per basic block for all basic blocks with more than two opcodes. This indicates that the majority of the basic blocks are dominated by a small value of k, and *5* is an appropriate choice for this dataset.

Choosing an Appropriate Bitvector Size. The bitvector size m strikes the trade-off between efficiency (similarity) computation and approximation error of using bitvectors

[6] A distance of 1 indicates no similarity where a distance of 0 indicates identical similarity.

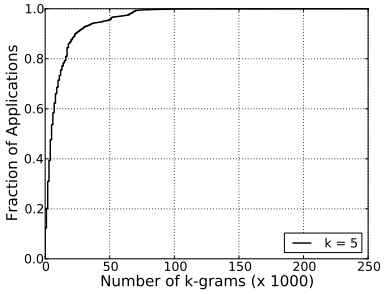

Fig. 9. Cumulative distribution of the number of unique k-grams extracted from 30,000 Android applications (with $k = 5$).

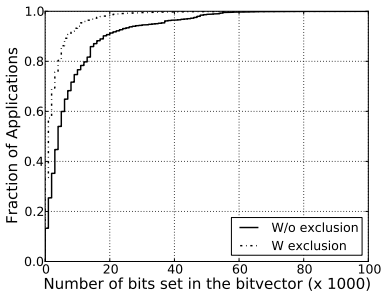

Fig. 10. Cumulative distribution of the number of bits set in the bitvectors of 30,000 Android applications.

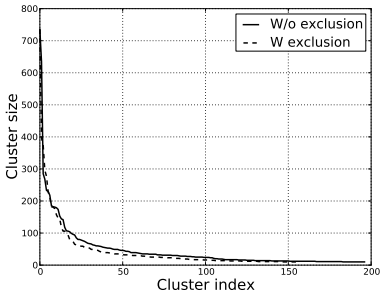

Fig. 11. The clusters obtained from 30,000 Android application using hierarchical clustering with similarity threshold $t = 0.9$.

to represent the sets of k-gram for each application. Ideally, we want size m to be large enough such that few collision occur during feature hashing. Practically, we want size m to be small so that we can efficiently compute similarity between all pairs of hundreds of thousands of applications.

According to [9], we need $m \gg N$, the number of k-grams extracted from an application, so that the Jaccard similarity between two bitvectors is very close to the exact representation of computing the intersection between two k-gram feature sets. In

addition, in all of our analysis, we are particularly interested in applications with high similarity, e.g., application pairs with similarity greater than 50%. Determining m by ignoring certain outlier applications will still yield good similarity results because those excluded applications are very unlikely to have a similarity score greater than 50% with other applications in the dataset.

We use all of the 30,000 applications from the Android Market to determine m. We compute the number of unique 5-gram features that can be extracted from each application, and plot its cumulative distribution from all applications in Figure 9. We find N_{90} in the distribution, which represents the threshold in which 90% of all applications' k-gram features are less than this value. We then set $m = 240,007$, a prime that is more than nine times of N_{90}, satisfying the condition $m \gg N$ suggested by [9].

We use the following two ways to verify whether $m = 240,007$ is large enough. 1) We do feature hashing with $m = 240,007$ for all 30,000 applications, and count the number of bits set in the bitvector for each application. We plot its distribution in Figure 10. We observe that more than 95% of applications have $1/5$ of their bits set in their bitvectors, and more than 90% of applications have only $1/10$ of their bits set. Hence, we do obtain sparse bitvector representation for the majority of applications. 2) We also randomly sample a subset of 1000 applications, compute the pairwise similarity among them using their k-gram feature sets, and compare the similarity values to those computed using their bitvectors. We find that the average difference between them is less than 0.01. With these two observations, we conclude that $m = 240,007$ is suitable for our analysis.

Clustering for Application Topology We use clustering as a way to group similar applications together. We run hierarchical clustering on the 30,000 Android applications using a similarity threshold $t = 0.9$, with and without a core functionality exclusion list applied, respectively. Figure 11 shows the cluster size sorted in a descending order. We see that both versions of the clustering worked well, but clusters with exclusion no longer had application clusters dominated by large libraries. These results show that clusters were not dominated by larger, uninvoked libraries that are present in many applications.

We observe that there are around 200 clusters, each of which has at least 10 applications, and in total there are 9344 applications in those clusters. Looking into the detail of the clusters, we find that our clustering identified three unique, commonly occurring patterns. They are:

Same application title, different versions. One cluster contained several versions of the same movie player, which were all responsible for displaying elicit pictures of models. Within the cluster, there were *4* different versions of the same model's movie player[23].

Differing author and functionality, same tool for development. In one example, AppBar[24] is a tool for allowing users to visually create applications for Android without needing to know about the underlying development platform. The platform allows for the addition of sounds, images, twitter feeds, and all sorts of additional widgets. Using our analysis, we identified a cluster on the official Android market consisting

of *735* applications of this type, ranging from RSS feeds to audio programs. It is the largest cluster we found in the applications from the official Android market.

Multiple apps from an author, different underlying functionality. A common pattern is for a developer to make a framework for creating applications and then reusing the applications in a variety of contexts. For instance, the company BrightAI[25] produces a variety of applications related to sports, from cycling to keeping tabs on favorite sports teams. One such cluster contained *28* different applications, all by BrightAI, but with different application purposes.

5.5 Case Studies

Previous work on studying Android applications[2] has shown that developers copy and paste code snippets from popular programming web sites into their own code, without understanding the potential security risks posed by blindly copying code.

Recently, Google announced an In-Application Billing API along with a sample application which demonstrates how the purchasing protocol works[26]. Several security warnings accompany the document, including statements regarding how developers should obfuscate their code, protect their purchasable content, and verify purchases on a remote server. However, the sample application ignores most of their suggestions in order to provide an easy to understand, boiler-plate application.

We show how Juxtapp can not only detect applications in the Android Market that copied this sample code, but we also show how we can detect other known source code-related vulnerabilities in the market using our architecture.

Reuse of Vulnerable Code In this section, we examine two cases of vulnerable code reuse of sample code provided by Google: In-Application Billing and the License Verification Library. We show that Juxtapp can quickly and efficiently reduce the set of potentially vulnerable applications and detect vulnerable code reuse in Android Applications.

In-Application Billing. Google In-Application Billing (IAB) is a library provided for developers to include so that their customers can sell digital content within their application, while letting Google handle authentication and credit card purchases[5]. For security reasons, Google advises that developers use obfuscation in order to make the code more difficult to understand for an adversary and they also recommend that developers perform verification on a remote server.

However, the sample code provided by Google is not obfuscated and performs verification of a purchase on the device. The left side of Figure 12, Line 231, shows the potential single point of attack. Meaning, if a developer can rewrite the statement to negate the condition, or force it to be true in some other way, the application will skip verification and allow the current user access.

In order to detect a potential attack, we analyzed the containment between the IAB sample code and the 30,000 applications in our dataset. We set a threshold that at least 70% of the IAB sample code must be in the application before further exploration.

Running containment between the sample IAB code and the Android Market applications took *1.5 minutes*,and we detected **295** applications containing 70% of the IAB

code. Other researchers used these applications to demonstrate that they could use the tool they developed for application rewriting to automatically exploit a vulnerability to get virtual goods for free [27]. Of those that used a significant portion of the sample code, **174** were vulnerable, while 65 use off-device/JNI verification and 56 were inoperable after rewriting. Our results show that Juxtapp is a fast way to quickly analyze large sets of applications for vulnerabilities caused by code reuse.

License Verification Library. The License Verification Library(LVL) is a library provided by Google in order to allow developers to query the Android Market at runtime in order to determine if a user is licensed to use a particular application[4]. Similar to IAB, Google provides sample code which encourages developers to obscure their code and ensure that single points of attack are protected. The sample code uses caching in order to prevent having to contact the Android Market every time the user invokes the application. However, the right side of Figure 12, Line 133, shows the potential vulnerability. This line could be rewritten to negate the condition, or to check another condition, making this a single point of failure, allowing a clever attacker to use the library without a license.

We executed containment on 30,000 using the Google LVL sample code to guide the search. For this experiment, we detected 272 potential candidates, **182** of which had 90% of the code, and **90** more, with at least 70% of the sample code. It took about *2 minutes* to analyze the dataset. Of the potentially vulnerable candidates, **239** of the 272 applications had the vulnerable pattern in their code. We manually verified the results in order to be assured that the pattern was in the code. Our analysis took about 10 minutes with script assistance responsible for opening each document which allowing the analyst to determine if the pattern exists, without the task of manually opening each file. Of those detected, some had obfuscated class and method names, but Juxtapp was still able to detect similarity. A few of the applications which were not vulnerable omitted the check for a cached response so that each time the user wished to use the application, a license must be granted[7].

```
222: boolean verify(...) {
231: if (!sig.verify(
232:         Base64.decode(signature))){
233:     return false;
234: }
235: return true;
```

```
130: void checkAccess(...) {
131: // If we have a valid recent LICENSED
132: // response we can skip asking Market.
133: if (mPolicy.allowAccess()) {
135:     callback.allow();
136: } else {
137:     //verification code
```

Fig. 12. The code on the left shows the vulnerable code present in the In-Application Billing Example Code `Security.java`. On the right is the point of vulnerability within the License Verification Library sample code `LicenseChecker.cpp` .

[7] Note: While this code is technically safe, Google advises against this because applications will be unusable when a user does not have Internet access.

Malware	Instances Found	Distinct New Carriers Found	Malware BB Size
GoldDream	25	13	1,898
DroidKungFu	6	0	5,357
DroidKungFu2	2	0	375
zsone	1	0	280
DroidDream	0	0	2,526
Total	**34**	**13**	-

Table 2. Number of instances of each kind of malware found in the Anzhi Market dataset. Also shown are the distinct new carriers discovered in our dataset.

Android Malware The Android Market place has recently experienced an influx of malware. Google has responded by exercising its remote application removal ability, that is, if Google determines an application is malicious or untrustworthy, it can remotely push a command to remove the application from affected devices[28]. In fact, as of August 2011, users are 2.5 times more likely to encounter malware on their mobile devices than only 6 months ago, and it is estimated that as high as 1 million users have been exposed to mobile malware[6][29]. We suspect that unregulated, 3rd party markets will have a higher incidence of malware.

Containment between Anzhi Market and Malware. In order to evaluate whether third party markets contain known malware, we select a subset of 5 malware from our dataset, which represents some of the most prolific, well-known malware. They include: DroidDream, DroidKungFu1/2, zsone and GoldDream. Each malware sample had a manual exclusion list applied, that is, using widely available malware analysis, we excluded common code from malware such as advertising libraries and common utilities which contribute nothing to the uniqueness of the code, all the while being overly conservative and leaving any questionable code in the application. The exclusion list can also be generated in a semi-automatic manner. Juxtapp can simplify this task for the analyst by first attempting to find a matching carrier application which is the same as the malware and then removing all similar code between the matching applications. The resulting code would be an exclusion list with only the differences between the applications remaining, which in this case is the malware exclusion list.

Table 5.5 shows that we were able to detect 34 malware in the off-market dataset. The experiment took around 10 minutes to complete.

Among those that matched we noticed a very high incidence of code reuse ranging from 93%-100%. The lower percentage matching shows that the technique is amenable to code mutations and variants. When investigating those with lower percentages, we noted that variants often changed file paths, reworked small amounts of code, changed exploit names, etc. While those matching 100% indicated with high probability that the two pieces of malware are identical and indeed, when investigated the samples matched.

When evaluating the samples we also consider the ratio of the malware sample compared to the container application. A low ratio indicates similar orders of magnitude among the code sample, where a higher ratio indicates that the reported matching is likely a false positive due to the density of the bitvector representing the larger ap-

plication. For instance, when evaluating our dataset for containment of GoldDream, the highest percentage match outside of the range given above was 73% with a code ratio of 20 times greater than that of the sample, indicating a false positive. This sort of clear demarcation allows us to quickly and easily identify malware samples and discard false positives.

Some malware found in the Anzhi market matched our sample malware dataset with little variation in code between them. However, other matched malware was significantly different from our evaluation set and we show how we can detect new variants, with new malware carriers using Juxtapp.

Most of the minor changes were related to class and package names. However, we point out that each of these applications' APKs were distinct on the market, each having a different MD5 sum. However, Table 5.5 shows that we found 13 unique carriers of the GoldDream malware in our dataset. Meaning, of these we found **13** previously unknown to us, distinct applications in our evaluation dataset, which were mostly all different types of games that had been repackaged with the GoldDream malware. Of the 13 significant GoldDream variants, one of them was found 12 times in the Market with only very small differences between them, so we consider this to be one instance. The differences between very closely matching variants were mostly constant strings, relating to the storage of various dropped files that the malware uses, along with variation in the class and package name, but little else. Our results confirm our suspicion that third party markets house known malware and that Juxtapp can be used to find known malware and previously unknown, new variants of them.

Identifying Contaminated Code. As a further step in analysis and verification of malware, Juxtapp can identify the commonality between applications and exclude those features that are contained within both applications and output the resulting code. By excluding the code that caused the match, an analyst can quickly discover which portions of the code were modified or injected into the application. For instance, when conducting the GoldDream experiments above, identifying the carrier application is simplified because the malware portions of the application are removed. This allowed us to quickly and easily identify the main components of the applications.

Containment between Android Market and Malware. We set out to determine if known malware was currently on the Android Market. We evaluated containment between 63 malware samples to the 30,000 collected from the official Android Market. The experiment took 19 minutes to execute locally.

Juxtapp did not detect any instances of known malware on the Android Market. This result is unsurprising given that Google has been vigilant about removing malware once it is found, banning the associated account, and issuing remote removal[30].

However, as expected, Juxtapp was able to detect the original application that the malware sample had been repackaged with in order to trick users into downloading. That is, a subset of our samples were repackaged with legitimate applications. Table 3 shows the Android applications we were able to detect using the malware sample.

Piracy and Application Repackaging In addition to vulnerable code and malware on the Android markets, piracy, especially among games, has become a major problem

Application File Name	Features	Name	Repackaged with
com.codingcaveman.solotrial.apk	4,272/4,831	Guitar Solo Lite	DroidDream.1
it.medieval.blueftp.apk	19,597/18,946	Bluetooth File Transfer	DroidDream.2
com.tencent.qq.apk	28,712	Tencent QQ Messaging	PJApps
de.schaeuffelhut.android.openvpn.apk	2,009	OpenVPN Settings	DroidKungFu

Table 3. Juxtapp is able to detect the original (and versions) of the application which was repackaged when compared to our malware dataset. Multiple features indicate multiple versions in our dataset.

for developers. Android applications are often pirated by rogue authors, which remove copy protection and replace developer revenue mechanisms with those that support the pirate (such as advertising libraries). In order to examine the third party market Anzhi for piracy, we downloaded and paid for the two applications mentioned in the Guardian article about android privacy[7]: 1) Chillingo's The Wars; 2) Neolithic Software's Sinister Planet. We compared these applications against the 28,159 applications in the Anzhi market, which took around 19 minutes to execute locally.

We found no instances of the Sinister Planet program being pirated on a third party market. However, we found **3** pirated versions of Chillingo's The Wars, being marketed by the company Joy World, the same company accused of piracy in the article. Each of the pirated versions has **71%** code in common with the original application. Among these versions, two are distinctly different, and the third is just a minor variation (string differences in the application code).

Despite the fact that the legitimate Wars program is unobfuscated, the Joy World version is obfuscated with methods and classes renamed. Additionally, we found that the pirate had added libraries to the application which were not present in the original version. So, even in light of significant obfuscation and additional code added, we were still able to detect similarity showing that Juxtapp handles perturbations in code well. We found that advertising and other libraries like Youmi, Casee, Millennial Media, AdMob and Wooboo were added to the pirated versions to generate revenue for the pirate[31–35]. Finally, the pirate did not remove the company name of the original producer of The Wars. Notably, the original maker, Chillingo, releases names under Deluxeware as well and this name *remains* in all pirated versions of the code.

In this section, we have shown that Juxtapp works well in detecting code reuse in Android applications. Namely, we've shown that instances of piracy, buggy code reuse, and known malware, even with obfuscation, can be detected by our tool.

6 Related Work

For large-scale malware analysis, Jang *et al.*[9] developed *BitShred*, a system for large-scale malware triage and similarity detection based on feature hashing. However, they focus on the technique as a contribution and classify x86 malware, whereas we apply similar techniques, with domain specific knowledge in order to find a variety of code reuse in Android marketplaces. We thank the authors of BitShred paper for sharing us their implementation of hierarchical clustering. Bayer *et al.*[36] use locality sensitive

hashing (LSH)[37], a similar but complementary technique, to increase scalability in malware comparison. Instead of using boolean features, the Zynamics BinDiff tool[38], Gao *et al.*[39] and Hu *et al.*[40] use features based upon isomorphisms between control flow and function call graphs of the program. Although the similarity computation based on graph isomorphism is expensive, it is less susceptible to being fooled by polymorphism. While these work primarily focus on techniques to compare and index malware, our work is focused on techniques to determine similarity among Android applications, and conducting much deeper security analysis to detect code reuse, malware contamination, and application repackaging in these applications.

Winnowing, a fuzzing hashing technique that selects a subset of features from a program for analysis, has been widely used for code similarity analysis[41] and plagiarism detection[42]. However, the winnowing algorithm requires calculating set inclusion, which is expensive when comparing many features. Additional plagiarism detection techniques are explored in [43–45], but they use complimentary techniques for plagiarism detection. Namely, source code clustering and manual analysis, program dependency graphs, and measuring approximate Kolmogorov complexity between programs, respectively.

A variety of approaches for static code clone detection have been proposed in the programming language literature for refactoring, finding bugs, and better understanding of the code. They roughly fall into two categories: syntactic level analyses and semantic level analyses. Semantic-based approaches[46–48] aim to reason about the program on the semantic level, such as using program dependency graphs (PDGs) or program slicing. This type of detection is very resilient to syntactical modifications; however, they are generally expensive and unscalable to large-scale analysis. In comparison, syntactic-based detection, including string-based[49, 50], token-based[51, 52] and tree-based[53, 54] tools, are much more scalable, but at the same time are much more vulnerable to syntactical modifications. All those techniques can be applied into our framework to further improve the accuracy and robustness our approach.

7 Conclusion

In this paper we presented Juxtapp, a scalable architecture for detecting code reuse in Android applications. Our architecture is implemented in Hadoop and we ran it on Amazon EC2. We evaluated the efficacy of Juxtapp in detecting vulnerable code reuse, known malware, and piracy in a dataset of 58,000 applications from Android marketplaces. In the evaluation of vulnerable code reuse, we found that many developers did not modify the sample code significantly from the Google-provided libraries, which left applications vulnerable to a variety of rewriting attacks. Furthermore, we used Juxtapp to discover **34** instances of malware, including *13* variants of the GoldDream malware which we found to be using a variety of games as carrier applications. Finally, we found *3* instances of piracy in which a game was victim to removal of copy protection and the addition code to include a multitude of advertising libraries which benefit the pirate. All these findings show that Juxtapp is a valuable architecture in detecting application similarity and code reuse in Android applications.

References

1. Yarow, J., Terbush, J.: Android is totally blowing away the competition http://www.businessinsider.com/chart-of-the-day-android-is-taking-over-the-smartphone-market-2011-11.
2. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of ACM CCS. (2011)
3. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of MobiSys. (2011)
4. : Google license verification library http://developer.android.com/guide/publishing/licensing.html.
5. : Google in-app billing http://developer.android.com/guide/market/billing/index.html.
6. : Mobile threat report https://www.mylookout.com/mobile-threat-report/.
7. : Developers express concern over pirated games on android market http://www.guardian.co.uk/technology/blog/2011/mar/17/android-market-pirated-games-concerns/.
8. KilianWeinberger, Dasgupta, A., Langford, J., Smola, A., Attenberg, J.: Feature hashing for large scale multitask learning. In: Proceedings of ICML. (June 2009)
9. Jang, J., Brumley, D., Venkataraman, S.: Bitshred: Feature hashing malware for scalable triage and semantic analysis. In: Proceedings of ACM CCS. (2011)
10. : Anzhi android market http://www.anzhi.com/.
11. : Number of available android applications http://www.appbrain.com/stats/number-of-android-apps/.
12. : Dalvik virtual machine http://www.dalvikvm.com/.
13. : Proguard http://developer.android.com/guide/developing/tools/proguard.html.
14. Walenstein, A., Lakhotia, A.: The software similarity problem in malware analysis. In: Proceedings of Duplication, Redundancy, and Similarity in Software. (2007)
15. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. Journal of Machine Learning Research **7** (December 2006)
16. Shi, Q., Petterson, J., Dror, G., Langford, J., Smola, A., Strehl, A., Vishwanathan, V.: Hash kernels. In: Proceedings of AISTATS'09. (2009)
17. : Hash functions http://www.cse.yorku.ca/~oz/hash.html.
18. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification. John Wiley and Sons (2000)
19. : Hadoop http://hadoop.apache.org/.
20. Sahoo, N., Callan, J., Krishnan, R., Duncan, G., Padman, R.: Incremental hierarchical clustering of text documents. In: In Proceedings of CIKM. (2006)
21. Gurrutxaga, I., Arbelaitz, O., Martn, J.I., Muguerza, J., Prez, J.M., Perona, I.: Sihc: A stable incremental hierarchical clustering algorithm. In: In Proceedings of ICEIS. (2009)
22. : Contagio malware dump http://contagiodump.blogspot.com/.
23. : Movie player heaven8 https://market.android.com/developer?pub=heaven8.
24. : Appsbar http://www.appsbar.com/.
25. : BrightAI http://www.brightai.net/.
26. : In-app billing http://developer.android.com/guide/market/billing/index.html.
27. : Freemarket: Shopping for free in android applications. In: Extended Abstract, to appear NDSS. (2012)

28. : Exercising our remote application removal feature http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html.
29. : Up to a million android users affected by malware, says report http://www.linuxfordevices.com/c/a/News/Lookout-malware-report-2011/.
30. : Update: Security alert: Droiddream malware found in official android market http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream/.
31. : Youmi advertizing http://youmi.net.
32. : Casee advertising http://www.casee.cn.
33. : Millennial media http://www.millennialmedia.com/.
34. : Admob http://www.admob.com/.
35. : Wooboo http://www.wooboo.com.cn/.
36. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: Proceedings of NDSS. (2009)
37. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Communications of the ACM **51**(1) (2008)
38. : zynamics bindiff http://www.zynamics.com/bindiff.html.
39. Gao, D., Reiter, M.K., Song, D.: Binhunt: Automatically finding semantic differences in binary programs. In: Proceedings of the 4th International Conference on Information Systems Security. (2008)
40. Hu, X., cker Chiueh, T., Shin, K.G.: Large-scale malware indexing using function call graphs. In: Proceedings ACM CCS. (2009)
41. Baker, B.S., Manber, U.: Deducing similarities in java sources from bytecodes. In: Proceedings of the USENIX Annual Technical Conference. (1998)
42. Schleimer, S., Wilkerson, D., Aiken, A.: Winnowing: Local algorithms for document fingerprinting. In: Proceedings of the ACM SIGMOD/PODS Conference
43. Moussiades, L., Vakali, A.: Pdetect: A clustering approach for detecting plagiarism in source code datasets. Comput. J. **48** (November 2005) 651–661
44. Liu, C., Chen, C., Han, J., Yu, P.S.: Gplag: detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. KDD '06, New York, NY, USA, ACM (2006) 872–881
45. Chen, X., Francia, B., Li, M., McKinnon, B., Seker, A.: Shared information and program plagiarism detection. IEEE Transactions on Information Theory (2004) 1545–1551
46. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Proceedings of the 8th International Symposium on Static Analysis. (2001)
47. Gabel, M., Jiang, L., Su, Z.: Scalable detection of semantic clones. In: Proceedings of the 30th international conference on Software engineering. ICSE '08, New York, NY, USA, ACM (2008) 321–330
48. Kim, H., Jung, Y., Kim, S., Yi, K.: Mecc: memory comparison-based clone detector. In: Proceeding of the 33rd International Conference on Software Engineering. ICSE '11, New York, NY, USA, ACM (2011) 301–310
49. Baker, B.: On finding duplication and near-duplication in large software systems. In: Reverse Engineering, 1995., Proceedings of 2nd Working Conference on. (jul 1995) 86 –95
50. Baker, B.S., Baker, B.S.: Parameterized duplication in strings: Algorithms and an application to software maintenance. SIAM Journal on Computing **26** (1997) 1343–1362
51. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7) (July 2002)
52. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering **32**(3) (2006)

53. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of ICSE. (2007)
54. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. Proceedings of International Conference on Software Maintenance (1998)