Juxtapp and DStruct: Detection of Similarity Among Android Applications

Saung Li

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-111
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-111.html

May 11, 2012

Acknowledgement

We conducted this research as part of Prof. Dawn Song's group dealing with the security of Android phones. We worked on Juxtapp in collaboration with Steve Hanna, Ling Huang, Edward Wu, Charles Chen, and Dawn Song. We would like to thank Steve Hanna and Ling Huang for providing numerous suggestions and advice for the DStruct approach and manually verifying that the cases of malware mentioned are indeed true positives. We framed parts of this report based on our paper on Juxtapp, and we used the Juxtapp workflow and incremental update figures from the Juxtapp paper. We would like to thank Edward Wu and Charles Chen for helping with the distributed implementation. We would also like to thank Prof. Vern Paxson for his helpful feedback on DStruct.

Juxtapp and DStruct: Detection of Similarity Among Android Applications

Saung Li

ABSTRACT

In recent years, we have witnessed an incredible growth in the adoption of smartphones, which has been accompanied by an influx of applications. Users can purchase or download applications for free onto their mobile phones from centralized application markets such as Google's Android Market and Amazon's third party market. Despite the rapidly increasing volume of applications available on the markets, these marketplaces often only cursorily review applications, and many applications are unreviewed due to the vast number of submissions. Markets largely rely on user policing and reporting to detect applications that may be misleading in its functionality or misbehaving. This reactive approach is neither scalable nor reliable as the incidence of piracy and malware has increased, putting too much responsibility on end users.

To automate the process of identifying problematic applications, we previously proposed Juxtapp, a scalable infrastructure for code similarity analysis among Android applications. Juxtapp is able to find instances of malware, piracy, and vulnerable code by detecting code reuse among applications. Such a system must be scalable and fast, so in this paper we discuss the distributed implementation of Juxtapp. We evaluate Juxtapp's performance on up to 95,000 Android applications and find that the parallelized system is able to analyze applications rapidly. To aid users in their analysis, we introduce a web service that automatically manages the resources that are required to run distributed Juxtapp, and we evaluate the performance of such a service.

For a complementary similarity analysis approach, we propose DStruct, a tool for detecting similar Android applications based on their directory structures. DStruct provides another method for performing similarity analysis to address problems in Android security, including determining if applications are pirated or contain instances of known malware. We evaluate our system using more than 58,000 Android applications from the official Android market and a Chinese third party market. In our experiments, DStruct is able to detect 3 pirated variants of a popular paid game and 9 instances of malicious applications on the Chinese market. Furthermore, on the official market, DStruct detected 4 legitimate applications that malicious authors had used to repackage with malware. We discuss the efficacy of DStruct and provide further insights into improving detection using similarity analysis tools such as ours.

1. INTRODUCTION

The increased popularity of Android phones, the open source nature of the Android operating system and development platform, and the availability of a wide range of devices compatible with Android have led to an explosive growth of the Android market share. As of January of 2012, Android holds 47% of the global smartphone market[1], and in 2011 Android's market share in China doubled from 33.6% to 68.4%[2]. Smartphone users can purchase or download applications onto their mobile phones from centralized software marketplaces. For

Android phones, Google hosts the official Android Market while Amazon and others provide third party markets.

The massive growth in the volume of Android applications has also led to increased occurrences of malware and piracy, which can hinder user experience and violate the intellectual property rights of application developers. As of August 2011, Android users are two and a half times more likely to encounter malware than six months ago and it is estimated that up to one million people were affected by Android malware in the first half of 2011[14]. Furthermore, application developers are concerned that a combination of lack of discoverability and ease of copying and republishing allows for frequent copyright infringement. One common case is for an illegitimate author to copy a paid or popular application, repackage it with added functionality, and rebrand it in order to generate revenue or execute malicious code[3].

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

One approach to addressing the problem of identifying misbehaving applications is to determine the similarity among applications. When an illegitimate author repackages an innocuous application with malware or creates a pirated copy of it, those two applications will likely contain similar features, such as having similar files or code. If the author creates a malicious application by repackaging and rebranding another malicious application, then those two applications will likely have similar features as well. Based on these observations, we explore two complementary techniques for performing similarity analysis on the applications' code and directory structures to detect instances of known malware in applications and to detect cases of piracy.

In our previous work, we introduced Juxtapp, a scalable architecture for quickly detecting code reuse and similarity in Android applications[4]. Juxtapp uses k-grams of opcode sequences of compiled applications along with feature hashing[35, 6] to represent applications efficiently and robustly. With this representation, we can compute pairwise similarity to detect code reuse among hundreds of thousands of applications. We previously demonstrated that Juxtapp is able to detect three different types of problems in applications: vulnerable code reuse, known malware, and piracy. One important goal of this system is to be able to quickly compare code and scale to large numbers of applications. In this paper, we explore in greater detail our distributed implementation of Juxtapp. We developed our distributed architecture using Apache Hadoop and ran it on Amazon EC2[42]. It is capable of fast, incremental additions to the analysis dataset so that it is amenable to frequent updates and additions to the pool of applications. We evaluate Juxtapp's performance and scalability using up to 95,000 applications collected from the official Android Market. We find that the system is fast, and we identify current bottlenecks inherent in the implementation. In addition, we introduce a web service that runs parallelized Juxtapp to allow for a fast and easy way to use our system and to enable the storing and sharing of analysis results. We walk through the stages of the web service to show how we prepare our data and resources for Juxtapp, and we evaluate the performance of these stages.

To demonstrate a new and complementary approach to Juxtapp, we also propose DStruct, a tool for determining the similarity among Android applications based on the directory structures of their archive formats. We convert the applications' directories and files into several tree formats to represent those applications. Using these representations, we have an effective method to compute pairwise similarity between applications to detect similar directory structures among a large dataset of applications. We apply DStruct to address the problems of piracy and known malware, and we evaluate DStruct's ability to detect these problems on 58,000 applications, which were collected from the official Android Market and the Anzhi third party market[15]. Using DStruct, we find 9 instances of known malware in the dataset and identify 3 pirated applications that are obfuscated with significant code changes from the original application. Furthermore, DStruct detected 4 legitimate applications that malicious authors had used to repackage with malware. We discuss the efficacy of DStruct and provide further insights into improving detection using similarity analysis tools such as Juxtapp and DStruct.

2. MOTIVATION

From December 2011 to April 2012 the number of applications in the Android market grew by about 100,000 to over 430,000[13]. The explosive growth of the Android market and the increased occurrences of pirated and malicious applications stress the need for a way to automatically analyze large numbers of applications for these problems. As such, we propose Juxtapp and DStruct, which are tools for automatically determining the similarity among Android applications based on their program code and directory structures to detect repackaged and pirated applications and known malware in the market.

Detecting code reuse and similar directory structures in Android applications offers a first chance in identifying applications that may negatively impact the user's security or defraud developers of revenue. This is an important step in improving the security of the markets as the problems of malware and piracy have risen dramatically. Android applications infected with malware increased from 80 in January 2011 to over 400 in June 2011[14]. Malicious authors often repackage legitimate applications with malware to mislead users into downloading infected applications. Meanwhile, pirates commonly repackage popular applications with modified code in order to evade copyright protection and generate revenue[3]. By comparing applications from the official Android market to a third party market we show that we can detect instances of piracy and known malware.

The constant influx of applications in recent years not only compels the need for fast, automated detection systems like Juxtapp and DStruct, but also requires an easy means for analysts to submit new applications for analysis, store and share the results, and query for any existing results. In light of this, we build a web service that accepts uploads of applications, runs parallelized Juxtapp to incrementally analyze and compare those applications against its existing dataset, and stores those results into a database. This service shares results from previous computations so that it prevents redundant work as analysts would not have to perform those same computations on their own. Instead, users can query from the database to quickly retrieve results of similarity comparisons among all analyzed applications. The service allows for easy usage of Juxtapp since it fully automates the transfer of data necessary for analysis, and the user does not need to know how to operate the system. This service provides a simple and efficient process for analyzing applications, which aids users in the early detection and notification of misbehaving applications.

The promising results of Juxtapp in detecting misbehaving applications using similarity analysis[4] motivates our exploration of complementary approaches to performing such analysis. As mentioned in Section 4.1, Juxtapp focuses on analyzing application code, whereas DStruct considers only directory structures. By examining a completely different aspect of applications, DStruct can reduce the number of false positives and uncover other cases of piracy and known malware. Using these two tools in conjunction would allow the analyst to detect misbehaving applications more quickly and makes evasion of detection more difficult.

There are several challenges that similarity analysis systems like Juxtapp and DStruct must address in order to work effectively. The system needs to perform comparisons quickly and scale to a large number of applications to meet the growth of the Android markets. It must accurately represent applications so that comparisons for similarity among them are meaningful and effective. The system needs to be resilient to certain levels of modification to the applications that generally occur when authors repackage applications. For example, authors may obfuscate the program code or add files to a directory in the application archive. The system must meet all of these goals while maintaining low false positive and false negative rates. For DStruct, there is an inherent tradeoff between false positive and false negative rates depending on how we represent the directory structure of applications, as we will discuss in Section 4.5.1.

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

In this section, we briefly summarize the approach of Juxtapp, describe its distributed implementation in greater detail, and conduct a performance evaluation. We then introduce our web service that automatically manages the resources necessary to run our distributed version of Juxtapp. For the interested readers, we encourage them to refer to our paper on Juxtapp for more details on the approach and our other evaluations on its effectiveness in detecting misbehaving applications[4].

3.1 APPROACH

As shown in Figure 1, Juxtapp consists of the following steps for analyzing Android applications: (1) application preprocessing, (2) feature extraction, and (3) similarity and containment analyses.

3.1.1 Application Preprocessing

Android applications are compiled from Java to Dalvik bytecode[36], and this compiled code is stored as the DEX format, which describes the application and retains class structure, function information, etc. The DEX file is stored in the application's archive format, called the APK. For each application APK, we extract its DEX file and convert it into a complete XML representation of the Dalvik program, including program structure. From the XML file, we extract each basic block and label it according to which package it came from within the application. For each basic block we only retain the opcodes and discard most operands. The result is a basic block (BB) file for each application.

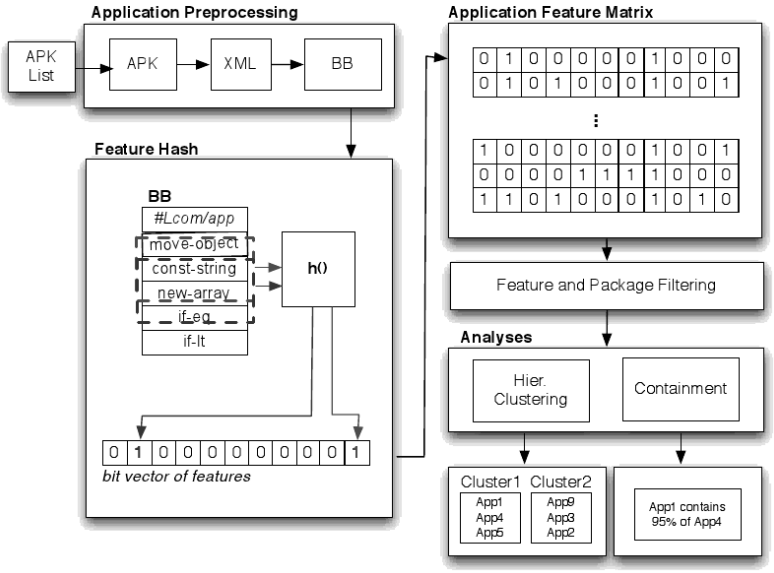

Figure 1: The Juxtapp Workflow.

3.1.2 Feature Extraction

We use k-grams of opcodes and feature hashing[35, 6] to extract features from applications. The k-grams extracted from code sequences effectively represent applications, and we feature hash them to reduce the dimensionality of the data. From each BB file, we extract k-grams using a moving window of size k and hash them. We ignore k-grams across basic blocks. For each hashed value, we set the corresponding bit in a bitvector, which represents the features in the application, to indicate the existence of the k-gram. These bitvector representations of applications, generated using feature hashing, are succinct and allow for efficient pairwise comparison.

3.1.3 Analysis of Feature Hashing Results

Similarity. We consider two applications to be similar if their bitvector representations of k-grams are similar. We calculate the Jaccard similarity which divides the number of features in common between applications by the total number of distinct features the applications have. More formally, the Jaccard similarity between bitvectors A and B is defined as $J(A,B) = |A \wedge B|/|A \vee B|$, and this value represents the similarity between applications. Using this similarity metric, we can perform agglomerative hierarchical clustering[39] on all of the bitvectors to group similar applications together into clusters.

Containment. We also perform containment analysis on the bitvectors to determine the percentage of code in common among applications. Containment is defined as the percentage of features in application A that exist within application B. We compute this value by dividing the number of features common in both applications by the number of bits in A. More formally, containment $C(A|B) = |A \wedge B|/|A|$.

3.2 DISTRIBUTED IMPLEMENTATION

In this section, we discuss our distributed implementation of Juxtapp, which also supports incremental updates to our application repository to scale with the growth of the Android markets. Juxtapp consists of 6,400 lines of C++, 1,600 lines of Java, and 600 lines of scripts.

We use the Hadoop MapReduce framework for managing computer nodes to perform large-scale data analysis computations and HDFS for sharing common data among nodes[40]. A single computer would require an enormous amount of time to run Juxtapp's techniques on all existing Android applications. This is infeasible as we need to conduct these analyses in a timely fashion to improve the security of applications available on the Android marketplaces. By distributing the work among many computers, we can significantly reduce the computing time as we increase the number of computers.

We wrote a MapReduce application that performs the application preprocessing step, and for the other stages we used Hadoop Streaming to interface with our C++ applications. Many of the tasks that Juxtapp requires are easily parallelizable, which allows us to greatly improve performance when dealing with large datasets. As a result, Juxtapp can feature hash and conduct similarity analysis in a distributed manner which offers great performance boosts over a single machine implementation.

3.2.1 MapReduce

In MapReduce, functions are defined with respect to key-value pairs. We use MapReduce by implementing two functions: a map function and a reduce function. The map function takes in key-value pairs and generates an intermediate set of key-value pairs. The reduce function takes in these intermediate pairs and merges all values that are associated with the same intermediate key. Using these map and reduce functions, MapReduce automatically parallelizes the work on a cluster of machines. In particular, one machine, the master node, partitions the large dataset into smaller pieces and distributes them to worker (or slave) machines. These worker machines then parse key-value pairs out of the input data and call the map function with those pairs as arguments. The map function returns intermediate key-value pairs, and the master node next assigns these pairs to reduce workers. These reduce workers read the intermediate key-value pairs assigned to them and groups all the values with the same key together. For each key and list of values, the reduce workers pass them as arguments to the reduce function, and the output of the function is appended to a final output file in the user program. By distributing work in this fashion to many machines at once, MapReduce can significantly reduce the processing time for a large data set.

3.2.1.1 Hadoop Framework

For our system, we use Hadoop, which is an implementation of the MapReduce framework. Hadoop MapReduce applications are written in Java, so for our C++ applications we use the Hadoop Streaming utility, which creates map and reduce jobs with any executable as the mapper or reducer. Both the mapper and the reducer are executables that read the input from `stdin` and emit the output to `stdout`. We note that Hadoop Streaming is unable to take advantage of Hadoop MapReduce's automatic resource management due to the externalities of the program being interfaced, as we will discuss later, but we find our tasks to still be easily

parallelizable. We also make use of Hadoop's own file system called the Hadoop Distributed File System (HDFS), which creates many replicas of data blocks and distributes them to cluster machines to enable reliable and rapid computations.

3.2.2 Application Preprocessing

The first step in the process is to convert the Android application file (APK) to a format that our architecture uses. The conversion step for each application can be done independently so the distribution of work is straightforward, and we write a Hadoop MapReduce application to perform this step. We define the map function to take in an application name as the key and output the result of preprocessing that application. Each APK contains a `classes.dex` file which contains the Dalvik representation of the Java code of the application. Inside of the map function, we parse through the application's DEX file and convert it to an XML format with the functions split into basic blocks. Next, we parse through the XML file and output to a basic block (BB) file all the opcodes contained within a basic block along with a label, which includes the source package, class, and method names. Using this map function, we assign a subset of the applications to each map task, which converts them to the basic block format and stores the resulting BB files into HDFS. No reduce function is necessary for this step.

3.2.3 Feature Extraction

Our feature hashing application is written in C++, so to parallelize this step we use Hadoop Streaming, which requires us to manually manage the transfer of resources among the computer nodes to evenly distribute the work. We obtain a list of all the BB files that we generated and stored into HDFS in the previous step, and we evenly partition this list based on the map task capacity. The map task capacity is the number of nodes times the number of worker threads per node. More specifically, we divide the list into even parts, where the number of parts equals the map task capacity, and the number of items per part is the total number of BB files divided by the map task capacity. We use these partial lists as inputs to the map function.

As previously mentioned, Hadoop Streaming's map functions use lines from `stdin` as input. We define our map function in this stage to take as input the locations of BB files stored on HDFS. The map function transfers these BB files from HDFS to the local worker node and feature hashes them, outputting to feature hash files the feature vectors representing the applications. Each feature hash file stores the size of the bitvector, the number of entries that are set to one, and a list of the positions in the bitvector that are set to one. This list of positions essentially represents the features that appear in the program. With this map function, each map task takes in a list of BB files, feature hashes them, and stores the feature hash files into HDFS.

Since we evenly partitioned the list of BB files based on the map task capacity and we use these parts as inputs to the map function, each map task receives an even number of BB files to feature hash. We expect this distribution process to evenly divide the workload among nodes on average. There may be some BB files that are unusually small or large that may cause uneven workloads, but feature hashing is fast regardless and these cases are rare.

3.2.4 Similarity and Containment Analysis

After generating all of the bitvector representations of applications, we calculate a pairwise similarity matrix between all applications. That is, for each application we compute its similarity to all other applications. The map function in this stage takes as input rows of the similarity matrix to compute and outputs the resulting similarity values. Again, each row represents the similarity comparisons of one application against all applications. We divide up the rows and evenly assign them to map tasks. The number of rows that each map task receives is the total number of rows divided by the map task capacity. When running the Hadoop job, the map tasks store the resulting similarity computations onto HDFS.

Since each map task requires the bitvectors of all applications to perform the similarity computations, we must distribute all of the feature hash files to all of the worker nodes. Such distribution is manual as we cannot take advantage of Hadoop's resource allocation using Hadoop Streaming. There are a few ways to distribute files to worker nodes, as we discuss in Section 3.3.3. For smaller clusters, we simply have all of the worker nodes retrieve the feature hash files from HDFS in parallel.

When comparing two applications *A* and *B*, the result is the same whether we compute the similarity of *A* to *B* or *B* to *A*. As such, as an optimization we can reduce the number of computations in half if we only calculate the upper right side of the similarity matrix. However, we would not obtain significant performance gains by partitioning the rows to map tasks because some map tasks would receive rows with many similarity values to compute and other tasks would receive rows with few computations. Instead, since the comparisons are independent, we simply divide up the pairs of applications that we want to compare evenly among the map tasks. Each map task receives the same number of pairs, and the total number of pairs to compare is half as many as before as we do not compare the same pair twice anymore.

As the last step, Juxtapp also computes the containment between sets of applications using their bitvector representations. Given two bitvectors, the containment tool determines what features are common between applications and outputs the percentage of code in common. For our distributed implementation, we use the same approach as in computing the pairwise similarity matrix, except we are computing the pairwise containment matrix. The difference is that we cannot make the optimization where we reduce the number of pairs to compare by half. The reason is that the percentage of code that application *A* contains of application *B* may be different from the percentage that *B* contains of *A*, so we must compute the entire containment matrix. Note that since all of the worker nodes require feature hash files for both similarity and containment analysis, we transfer these files to all worker nodes from HDFS only once so that we can use them in both analyses.

3.2.5 Incremental Update

The many stages of Juxtapp are stateless, making it easy to incrementally process new applications, update their similarity and containment matrices, and analyze them in detail without having to reprocess all applications in the repository. Support for incremental updates is important because there are hundreds of thousands of applications on the markets to analyze and authors constantly submit new applications. We do not want to re-analyze the applications that we have already processed when we add new applications to the dataset, and we only want to perform the minimal computations necessary to update our analyses.

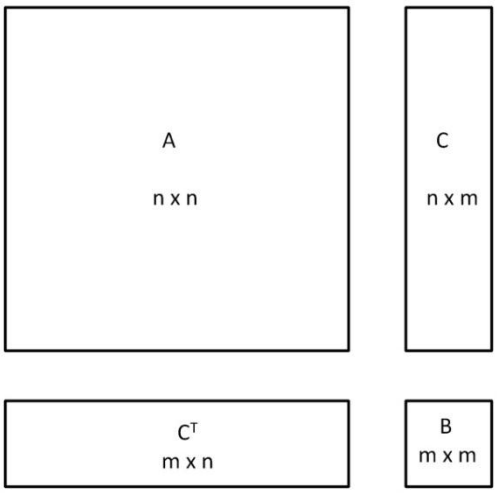

Figure 2: Incremental updates for the similarity matrix. *A* contains the similarity values among existing applications, *B* contains similarity values among new applications, and *C* contains similarity values between the new applications and the existing ones.

The application preprocessing and feature hashing stages are inherently incremental, meaning that when we add new applications to the analysis, we only need to preprocess and feature hash those new applications. These stages scale linearly with the number of application, so given *m* new applications to analyze, we have *O(m)* amount of work. For the similarity and containment analysis, we perform pairwise comparison among the new applications, and then for each new application we compare it to all existing applications. In other words, as shown in Figure 2, with *n* existing applications and *m* new applications, updating the existing *n x n* similarity matrix *A* is as follows: (1) compute *m x m* similarity matrix *B* among the new applications, (2) compute *n x m* similarity matrix *C* between the set of new applications and the existing ones, and (3) concatenate the results together and grow matrix *A* at appropriate rows and columns to get the new *(n + m) x (n + m)* similarity matrix. Updating the containment matrix is similar except we have to compute the *m x n* matrix as well. For the analysis stage, the the amount of work will be *O(m^2) + O(nm)*. As shown, we avoid having to process the existing applications again, which can take *O(n^2)* work. In practice, incremental processing greatly saves computation time since the number of existing of applications is large but the number of new applications we want to process is much smaller.

To support the incremental updates, we do not have to modify any of the Hadoop map functions. We simply divide up the incremental work into parts and assign them to the different map tasks using the same methods as previously discussed. For preprocessing and feature hashing, we assign a subset of the new applications to each map task. For the similarity and containment analysis, we assign a subset of the incremental rows we need to compute to each map task and join the results with the existing similarity and containment matrices at the appropriate rows and columns. Using this approach, we are able to distribute the incremental analysis among many machines. As before, we face the same bottleneck dealing with Hadoop Streaming where we must transfer the feature hash files of the existing and new applications to the worker nodes in order for them to calculate the incremental comparisons.

3.3 PERFORMANCE EVALUATION

In this section, we evaluate the performance of Juxtapp. We introduce the evaluation dataset, describe our experimental setup, and run performance experiments.

3.3.1 Experimental Evaluation Dataset

For our evaluation experiments, we collected Android applications from three different sources. From the official Android Market, we downloaded 30,000 free applications. Additionally, we obtained 28,159 free applications from Anzhi, a third party Chinese market[15]. Lastly, for one of the performance experiments, we use a set of 95,000 Android applications that we obtained from the official Android Market.

To gain a general understanding of our dataset, we calculated the average and total size of 30,000 unique applications as a representative sample from the official Android Market. The average APK file size from this set is 724KB, and the total file size of these APKs is 50.43GB. The average size of their feature hash files is 56.9KB, and the total size of all feature hash files is 1.7GB. The large discrepancy in sizes between APKs and feature hash files is due to the compact and efficient bitvector representations of applications.

3.3.2 Experimental Setup

For performing large scale experiments such as containment between on-market and off-market applications and generating a large pairwise similarity matrix, we applied our Hadoop implementation to Amazon EC2. We used two types of computer nodes for our Amazon EC2 clusters to conduct different experiments: m2.4xlarge and c1.xlarge instances. Our m2.4xlarge instances run on 64-bit Ubuntu Linux 2.6.38-8-virtual with 8 virtual cores, which each have 3.25 EC2 Compute Units, and 68.4GB of memory. The c1.xlarge instances are similar except they have 7GB of memory and each core has 2.5 EC2 Compute Units, meaning they have less processing power than the m2.4xlarge instances.

3.3.3 Distributing Files to Worker Nodes

Currently, we have explored three approaches for distributing files to workers. The first method is to `ssh` into all worker nodes and have them retrieve the necessary files from HDFS in parallel. The second way is for the master node to retrieve the files from HDFS, convert them to an archive file, such as a tar file, and send that archive as part of the Hadoop Streaming job using the `-archives` option, which copies the files to the workers[47]. The third way is to have the master node `rsync` the archive to all of the workers. This third approach works well for small numbers of nodes, but it is not scalable to the cluster size because every worker node is copying from the master and can overload it. Using `rsync` to transfer an archive of 30,000 feature hash files to 10 instances of c1.xlarge worker nodes takes 4 minutes. To transfer that same archive to a cluster of 80 nodes, it takes 75 minutes. On the other hand, transferring files using `-archives` is fast and scalable, as it takes under 4 minutes to transfer the same archive to 80 workers. However, we must first transfer the files from HDFS to the master, which then archives them,

before using the `-archives` option in the Hadoop job. Converting 30,000 feature hash files to a tar file takes 10 minutes. The first method takes 7 minutes in total to transfer the same number of feature hash files to 20 nodes, and about 10 minutes to 80 nodes, making it the best choice out of the three approaches for this many files. However, as the number of feature hash files in the dataset increases, the distribution overhead will increase as well since we must transfer all of those files to workers. The `-archives` option is known to work better for large clusters than the first approach when the archive sizes are much larger[47]. When there are too many machines trying to retrieve many small files from HDFS, they must hop from node to node to search for the required files, resulting in communication overheads. Hadoop works better when working with larger but fewer files, and we revisit the problem of small files in Section 3.4.2.

One possible approach to reduce the parallelization overhead is to write Java Hadoop MapReduce applications for our feature hashing, similarity analysis, and containment stages instead of using Hadoop Streaming. By doing so, we can take advantage of Hadoop's resource allocation management instead of having to manually transfer the required files onto the worker machines. Another improvement is to use `DistributedCache`, which caches files needed by applications so that files are only copied once per job, possibly speeding up the transfer of files to worker nodes[46]. We leave these approaches as part of future work.

3.3.4 Performance over a Large Dataset

One of our system goals is to perform comparisons fast, especially when we have a large number of applications. Using Amazon EC2, we can increase the number of worker nodes to make the time of computation over large datasets manageable. To understand how much of a performance gain we can achieve, we run Juxtapp on 95,000 unique Android applications using a variable number of m2.4xlarge nodes ranging from 25 to 100 with 8 worker threads per node. We use the `-archives` method to transfer files to the worker nodes since we are dealing with a large dataset of small files. At the time of writing, there are around 430,000 Android applications, so our dataset represents a large fraction of the markets. Figure 3 shows the time required to complete full runs of the entire pipeline, which include the APK to BB file conversion, feature hashing, and the pairwise similarity computations. In these experiments, we assumed that the applications are stored on HDFS at the start, though in practice we would have to transfer them to HDFS, requiring longer running times. We made this assumption here because we wanted to test the scalability of our Juxtapp techniques and not the transfer times of applications.

As we can see from Figure 3, we can reduce the running time by increasing the number of worker nodes. However, as we increase the number of nodes, the performance gains decrease as the overhead of parallelization gradually dominates the time to analyze the applications. The pairwise comparisons require the transfer of feature hash files from HDFS to all of the worker nodes, and this overhead takes up more of the total running time as the conversion and computation times decrease. This transfer time presents a current bottleneck on performance as we cannot speed it up with the number of nodes. The figure shows how the overhead of the pairwise comparison approaches a constant overhead as the number of nodes increases. On the other hand, the preprocessing and feature hashing stages do not require any synchronized state for parallelization, so these stages contributed to significant performance gains as the number of workers increased. Despite the overheads, we see that if we use 50 nodes, the time to process

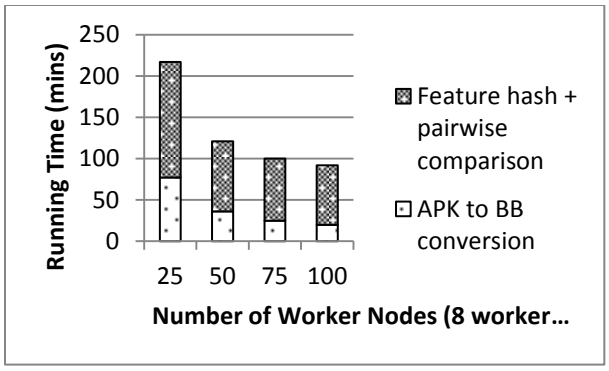

Figure 3: Time to preprocess, feature hash, and compare 95,000 unique Android applications using various numbers of workers per cluster on Amazon EC2.

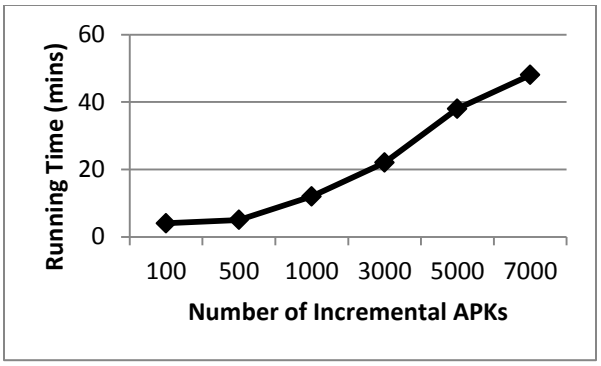

Figure 4: The time to incrementally process varying numbers of APKs using 20 worker nodes with 30,000 existing applications

95,000 Android applications is around 2 hours, which shows that parallelized Juxtapp can perform many computations very quickly. Given that this dataset is a large fraction of the total number of applications, Juxtapp scales well, and we demonstrate that we can perform our full analysis at a reasonable time using large Amazon EC2 clusters.

3.3.5 Incremental Update Performance

Incremental updates to the application dataset enables us to continuously process and update our dataset with new applications without requiring running Juxtapp on the whole application repository. We support this technique so that our system is able to scale with the growth of the Android markets. To assess the incremental performance, we conduct incremental update experiments using a cluster of 20 m2.4xlarge machines. We assume that we have already processed 30,000 applications and stored the results on HDFS, and we have a number of new applications on HDFS that we want to process. Figure 4 shows the times required to incrementally add from 100 to 7,000 APKs to the dataset using the entire workflow of Juxtapp, including preprocessing, feature hashing, similarity computations, and containment analysis. From the experiments, we see that as we increase the number of new applications to process, Juxtapp scales fairly well, almost linearly. As shown in Section 3.3.4, we can further reduce the running times using a larger cluster of machines when the number of new applications is large. However, for smaller numbers of applications, increasing the number of worker nodes will have less impact due to the overhead of parallelization. Our experimental results show that adding new applications to the dataset daily or even multiple times daily is feasible with Juxtapp.

3.4 WEB SERVICE FOR JUXTAPP

The rapid growth of the Android markets calls for a systematic way to process new applications, compare them against the existing dataset, and obtain the results. This entire process, including the management of resources and files, needs to be fully automated so that analysts can keep up with the growing number of applications and focus on the comparison results. In this section, we describe our web service, which provides an interface for users to automatically analyze applications using the distributed implementation of Juxtapp and store their results for later retrieval and further analysis.

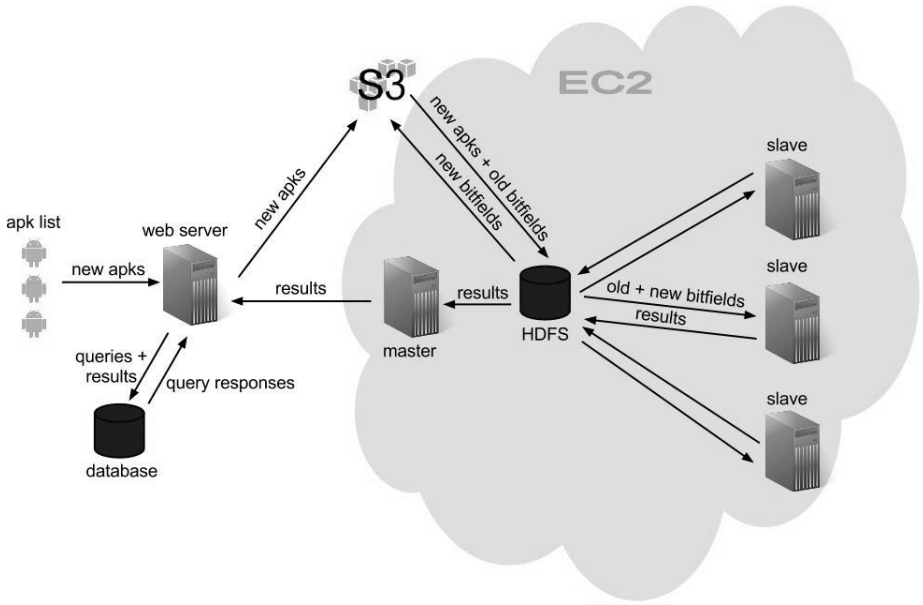

Figure 5: Overview of the Juxtapp web service. The arrows show the movement of data.

3.4.1 Approach

Figure 5 provides an overview of the various components of the web service and shows how the service transfers data among them. The web server enables two key user interactions through a website: the submission of new applications to run Juxtapp on and the queries for the results of previous analyses.

For new applications, the user uploads an archive of those applications to the web server, which then extracts and validates them. We do not want to store the applications on the web server, as they may overload it, so we transfer them to Amazon S3, which is a storage service on the web[41]. Transferring data from a local machine to S3 is time-consuming, so we immediately start the transfer of applications after extraction and run the process in the background. Since distributed Juxtapp currently uses Amazon EC2, which charges by the hour of usage, we do not want our clusters to always be running. As such, we only start a cluster of machines and process the applications stored on S3 once a day. By processing new submissions in batches, we use EC2 to perform more work over a shorter time period, which helps reduce costs. Once a day, the web server checks for the existence of submissions, and it launches and configures an EC2 cluster of machines. Next, we transfer the new applications as well as the feature hash files of already processed applications to the cluster's HDFS. To transfer these files, we use DistCp, which is a Hadoop tool that uses MapReduce to parallelize the copying using the cluster[43]. Once these files are stored on HDFS, we run distributed Juxtapp to incrementally process the batch of applications as explained in Section 3.2.5.

After running Juxtapp, we transfer the results from HDFS to the master node, which then parses and compacts the results. We also use DistCp to store the new feature hash files generated from Juxtapp to Amazon S3 for future incremental analysis. We send the processed results back to the web server, which then terminates the cluster while storing the end results into a database. To obtain the results of analyzed applications, users can submit query requests to the web server, which in turn queries from the database and outputs the results to the web page.

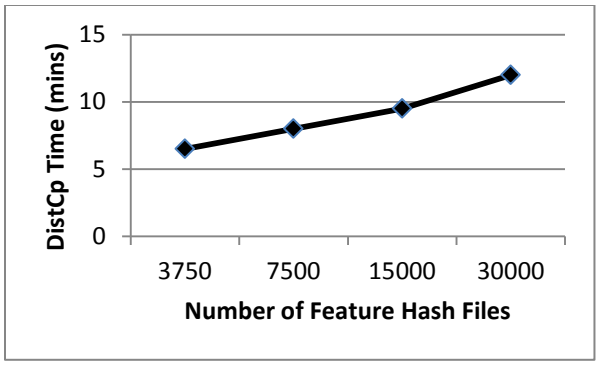

Figure 6: Time to DistCp varying numbers of feature hash files from Amazon S3 to HDFS using 20 worker nodes.

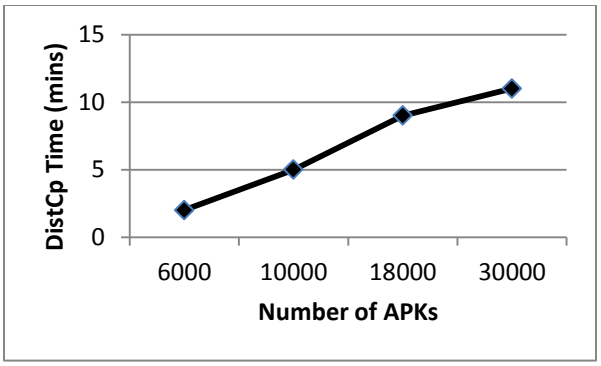

Figure 7: Time to DistCp varying numbers of APKs from Amazon S3 to HDFS using 20 worker nodes.

3.4.2 Performance Evaluation

Bottlenecks. As we have shown in Section 3.3, distributed Juxtapp performs fairly well in analyzing applications. However, the web service's movement of data before and after using Juxtapp presents numerous bottlenecks in the pipeline. In the initial step, the user uploads applications to the web server, and the speed will depend on the user's connection with the server and data transfer rates. Of course, uploading a larger number of applications will require a longer time to complete, and the scaling is linear. We store the uploads from the web server to S3, which poses another bottleneck since this step is also done serially and depends on the connection to S3. Storing 6,000 APKs to S3 from the web server takes around 4 hours to complete. These bottlenecks are manageable since we run the batch processing only once a day, and for the rest of the times we carry out these transfers of applications with negligible cost. We note that launching a cluster of machines to start a batch process can take a variable amount of time, as Amazon's EC2 service must fetch instances and set up their virtualization environment. It generally takes 20 minutes for us to launch a cluster of 20 c1.xlarge nodes and automatically configure them.

DistCp. In our performance evaluation of Juxtapp, we assumed that the new applications and feature hash files of old applications were already stored on HDFS. As part of the web service, we have to take into account the transfer of these files from S3 to HDFS as well as the transfer of new feature hash files to S3. To evaluate Hadoop DistCp's performance in transferring these files, we run experiments involving the transfer of a variable number of applications and feature hash files.

For Juxtapp's incremental processing, the web service transfers feature hash files between S3 and HDFS. Figure 6 shows the time required to DistCp a variable number of feature hash files from S3 to HDFS using 20 instances of c1.xlarge nodes (5 worker threads each). These feature hash files are from 30,000 applications from the official Android market. As mentioned in Section 3.3.1, the average file size is 56.9KB, and the total size of all these files is 1.7G. As we can see, the transfer time grows fairly linearly as the number of files increases, up to 12 minutes for 30,000 feature hash files, which is a reasonable time. In another experiment, we increased the number of nodes from 20 to 40, and also from 40 to 80, but we only saw decreases in transfer time by about a minute for 30,000 feature hash files for both cases. Larger clusters may split up the transfer work among more nodes to reduce the completion time, but the

overhead from bookkeeping and handling concurrent writes to HDFS neutralizes most of those gains. As such, we cannot improve this stage's speed using more nodes, which presents a bottleneck.

The web service also transfers new applications from S3 to HDFS to run Juxtapp for processing and analysis. Figure 7 shows the time required to DistCp a variable number of APKs using 20 c1.xlarge machines. We used 30,000 APKs from the official Android market. We note that the average file size is 724KB and the total size of these APKs is 50.43 GB. We see that these transfer times also grow fairly linearly with the number of APKs, and it takes about 11 minutes to transfer 30,000 APKs from S3 to HDFS. These running times are acceptable for our web service, though as in the previous experiment with feature hash files, we cannot significantly improve the performance of this stage using more nodes, which presents another bottleneck.

Interestingly, although APK file sizes are generally much larger than feature hash files, the DistCp times are slightly shorter for APKs than for feature hash files. This is due to the fact that HDFS is not geared to efficiently handle small files, as it is primarily designed for streaming access of large files[45]. In light of this, as a future improvement, we could combine many bitvectors into one file, while storing enough information to remember which application each bitvector corresponds to. By merging files into larger ones, we can reduce the number of seeks and hoppings from node to node to retrieve files. Furthermore, we reduce the number of map tasks for Hadoop jobs as there are less files to transfer, resulting in less bookkeeping for the MapReduce job. With this change, we would expect to see performance gains when using DistCp and in all stages of Juxtapp that use Hadoop to interact with our currently small feature hash files, as long as we do not introduce significant overhead to keep track of the combined bitvector files. Currently, we store all of Juxtapp's results onto HDFS and we transfer many feature hash files from HDFS to all worker nodes. The optimization could speed up all of these processes and reduce the bottlenecks.

Database. The performance of the interactions with the database depends on the database implementation. For our database, we use SQLite for storage of our simple results, and we experience good performance. SQLite can easily do 50,000 or more insert statements per second on an average desktop computer[44], so we can store our pairwise comparison results rapidly. If the database is stored on a machine different from the web server, then there would also be remote communication times added on top of the total running time.

Local Cluster. As an improvement, we can use a local cluster of machines instead of Amazon EC2 to eliminate parts of the web service's pipeline and significantly reduce the running time of a batch process. With our own continuously running cluster, we would not have to launch, configure, and terminate EC2 clusters for every batch process. We also would not need to use Amazon S3 for storage, since we could leave our results on the cluster's HDFS. This eliminates the steps that require the transfer of files to and from S3, resulting in significant performance gains. Furthermore, the master node could act as the web server so that we would not need to transfer the Juxtapp results back to the web server remotely. Users would submit new applications to the master node. Instead of sending the new applications to S3, the master node would transfer them to HDFS, which requires much less time than transferring files from a local machine to S3. Figure 8 shows the time to transfer various numbers of APKs to HDFS from the master node on a 20 node cluster. As we can see, the time to transfer 7000 APKs is just under an hour, as opposed to taking 4 hours to transfer 6000 APKs to S3 from a local machine. All of

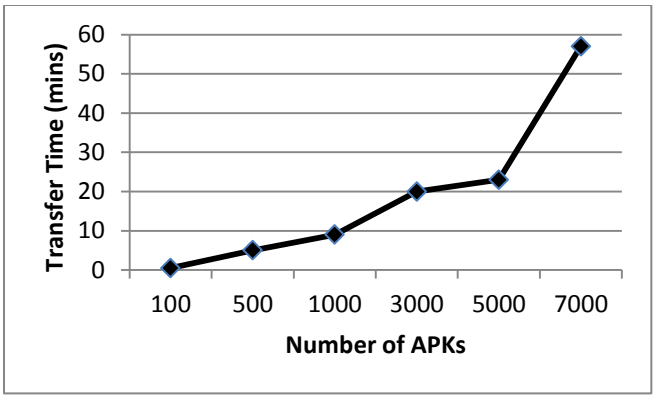

Figure 8: Time to transfer various numbers of APKs from the master node to HDFS on a 20 node cluster.

these speedups may not be necessary given that we are running a daily batch service, but by using a local cluster we could process submissions and return results to users faster.

4. *DStruct: A Tool for Detecting Similar Directory Structures Among Android Applications*

In this section, we propose a complementary approach to Juxtapp for performing similarity analysis to address problems in Android security, including piracy and known malware. We discuss related work and describe the general approach. We evaluate DStruct on more than 58,000 Android applications and detect instances of piracy and malware on the Android markets. We then discuss the limitations of DStruct and provide further insights into how to improve detection using similarity analysis tools such as DStruct and Juxtapp.

4.1 RELATED WORK

In our previous project, we developed Juxtapp, a scalable infrastructure for detecting code reuse in Android applications[4]. Like DStruct, Juxtapp computes similarity among applications to detect instances of piracy and known malware. However, Juxtapp focuses on program code and does not consider the directory structure of applications. It uses feature hashing on executable code sections to dramatically reduce the high-dimensional feature spaces that are common in code reuse analysis. Juxtapp faces a number of domain specific considerations when performing code similarity analysis: the source code is not available for most applications so we must include an involved preprocessing step; each application must include libraries that it is using, which leads to significant code copying that can undermine code similarity computations; Android developers often obfuscate code to make the reverse engineering of applications more difficult while misbehaving authors obfuscate code to evade detection. These problems motivate an alternative approach for determining the similarity between applications. DStruct is a simpler system that examines applications at a higher level and represents applications by their directory structures. This avoids all of the problems that are associated with the application code and provides another metric for determining application similarity. However, we do not consider DStruct as a replacement for Juxtapp, since the latter performs quite effectively and can detect cases of vulnerable code reuse, which DStruct does not address. Instead, we developed DStruct so that we could use it in conjunction with Juxtapp to

eliminate false positives and negatives and to reaffirm each other's results. In order for repackaged applications to evade detection, they must evade detection from both the program code approach and directory structure approach.

Zhou *et al.* developed a system for detecting repackaged applications using techniques similar to Juxtapp[5]. They focused mainly on detecting repackaged applications, while we also detect specific cases of both piracy and known malware using DStruct. For conducting large-scale malware analysis, Jang *et al*.[6] developed BitShred, a system for large-scale malware similarity analysis and clustering using automatic malware triage techniques. They focus on using feature hashing as a contribution and apply it on x86 malware, whereas we employ different techniques to detect repackaged applications in Android marketplaces. Bayer *et al.*[7] proposes a scalable clustering approach to group malware samples that exhibit similar behavior, and they perform dynamic analysis to obtain execution traces of malware programs. All of these approaches rely on analyzing program code, which we do not consider. Instead, we focus only on comparing directory structures of applications as a complementary approach to these code analysis systems.

Gao *et al*.[8] and the Zynamics BinDiff tool[9] use techniques for finding differences and similarities between programs based on isomorphisms between their control flow graphs. Computing similarity based on graph isomorphisms is expensive but is more resistant against polymorphism, which can fool other approaches. Meanwhile, Hu *et al*.[10] introduces a system for determining similarity of malware samples based on the malwares' function control graphs, which is a structural representation that is less susceptible to instruction-level obfuscations. Our analysis, on the other hand, involves comparing tree representations of directory structures instead of graphs generated from binary files or program code. As such, we analyze a different aspect of applications as a complementary approach, and our tree comparisons are less complex than computing graph similarity. These works primarily focus on introducing techniques to compare and index malware, but we focus on techniques to determine similarity among Android applications and conduct a deeper security analysis to detect malware repackaging and pirated applications.

We are currently unaware of previous research related to detecting similar applications based on their directory structures. Much of the work on similarity analysis has focused mainly on program code. However, like our system, current approaches typically involve constructing representations of applications and then comparing those representations as metrics for determining similarity[11].

Our work involves computing differences of trees, as we will explain in Section 4.3, and we specifically use a tree edit distance algorithm[29]. We use this algorithm because it is simple and works fast, but there are other algorithms that may perform more accurately or even faster. One approach involves approximating tree edit distances to improve computation speed at the expense of accuracy[28]. We can apply many other tree matching algorithms in place of the one we chose as they are complementary[30].

4.2 BACKGROUND

Application Directory Structure. Android applications are stored as APK files, which are an archive file format that consists of the applications' compiled code, certificates, and resources. Application resources include image files, database files, audio files, library files, XML files, and others. The files inside the APKs are typically organized into directories, and all of these files

```
> unzip -l BloodvsZombie_com.gamelio.DrawSlasher_1_1.0.1.apk
Name
-------
META-INF/MANIFEST.MF
META-INF/CERT.SF
META-INF/CERT.RSA
AndroidManifest.xml
res/drawable/icon.png
assets/data.pak
resources.arsc
classes.dex
lib/armeabi/libClawApp.so
-------
9 files
```

Figure 9: Files inside of a sample Android APK from the game BloodvsZombie.

```
ARC archive data, packed: 1
ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked,
stripped: 1
PNG image data, 64 x 64, 8-bit/color RGBA, non-interlaced: 1
Dalvik dex file version 035: 1
DBase 3 data file (5628 records): 1
data: 2
ASCII text, with CRLF line terminators: 2
```

Figure 10: A list of `file` data types and their counts from the BloodvsZombie game.

and directories make up the directory structure of the APK. In Figure 9, we use the unzip command to output a list of all the files in a small sample APK from a game called BloodvsZombie. All of the files and directories and their relationships among each other make up the directory structure of the application. Our approach relies on the observation that applications with different functionality typically have different directory structures, and applications that are similar, such as different versions of the same app, will have similar directory structures.

One way to examine the types of files inside of the APK is to run the Linux file command on them, which recognizes the types of data contained in them. The following are example usages on an image file and a database file:

```
> file sun.png
sun.png: PNG image data, 47 x 47, 8-bit/color RGBA, non-interlaced
> file AndroidManifest.xml
AndroidManifest.xml: DBase 3 data file (1976 records)
```

The output is of the format `filename: file type`, and for our purposes we extract only the file type when we run the `file` command. Figure 10 shows a list of file data types contained in the small sample APK from the BloodvsZombie game. We will discuss in Section 4.3.2.1 how we use this command as part of the representation of the directory structure.

Tree Representation and Comparison. There are several methods to represent the directory structures of applications so that we can apply comparison techniques to them. In our approach,

we naturally represent the directory structures as trees with labeled nodes, and once two applications are represented as trees, we can compute the tree edit distance between them. The distance between two trees is considered to be the number of edit operations to transform one tree to another. Edit operations include (1) changing one node label to another, (2) deleting a node, after which all children of the deleted node become children of that node's parent, (3) and inserting a node, after which a consecutive subsequence of siblings among the children of the node's parent become the children of the node. The greater the edit distance, the more different the two trees are from one another. There are many algorithms for computing tree edit distances, and in our approach we chose the Zhang-Shasha algorithm as it is simple and fast[29].

4.3 APPROACH

Our approach for DStruct involves the following two general steps for performing similarity analysis among Android applications: (1) constructing representations of the applications, (2) and calculating differences among the applications. DStruct constructs trees out of the directory structures and uses them as representations of the applications. To calculate differences among pairs of trees, we compute their tree edit distances. Figure 11 shows DStruct's workflow.

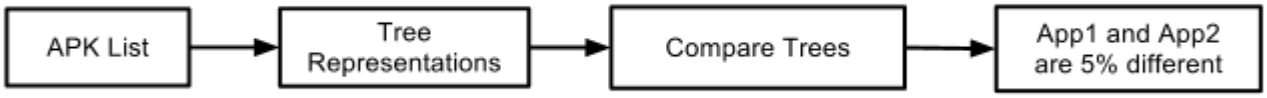

Figure 11: The DStruct workflow, which consists of converting APKs to trees, comparing the trees, and outputting their differences.

4.3.1 APK Extraction

APKs are ZIP file formatted packages based on the JAR file format, so we can use standard unzipping tools to extract the APKs. After extraction, we are free to access all of the directories and files of the applications.

4.3.2 Representation of Directory Structures

After extracting an APK, we walk through all of the directories and files of the application and construct a tree along the way to represent the directory structure. The leaves of the tree are the files and the non-leaf nodes are the directories of the application. A file leaf is a child of a directory node if the directory contains that file. The method for labeling nodes is crucial as it affects how closely the tree represents the directory structure of the application and, hence, influences the similarity computations among pairs of applications. We explore several methods for labeling nodes: using the names of the files, extracting the extensions of them, computing their md5sums, and using the Linux `file` command.

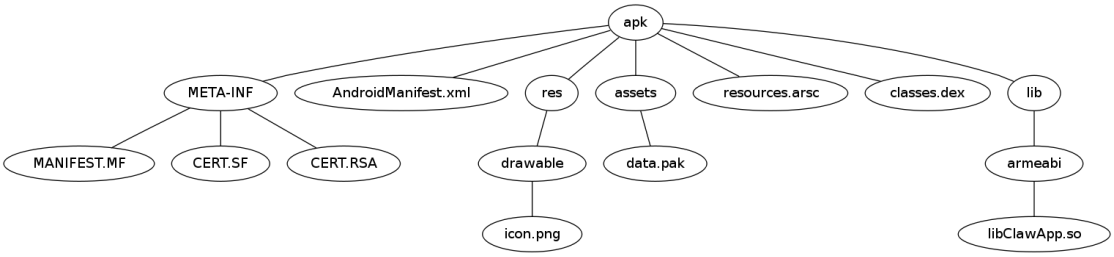

Figure 12: A tree representation of the directory structure of the BloodvsZombie game. We use the names of files and directories as the node labels in this example.

4.3.2.1 Labeling of Tree Nodes

One approach to labeling the nodes is to use the names of the files and directories. This leads to a simple representation of the directory structure, and with this labeling we consider applications to be similar if they have similar file and directory names. Figure 12 shows a tree representation of the directory structure of the BloodvsZombie game using the file names as labels. The problem with this method is that it is susceptible to the renaming of files and directories inside an APK when an author repackages the APK. If, for example, the author renames all of the files inside the repackaged APK, then that application will appear completely different from the original one, leading to a false negative.

A possible solution to the renaming problem is to label the nodes using the extensions of the files. For example, if the filename is `AndroidManifest.xml`, then the label will be ".xml". We label all directories as "directory" since they do not have file extensions. Renaming the file will have no impact on the representation since the extensions remain the same. However, the author could possibly rename the extension as well, leading to false negatives. For example, the author could change the filename `text.txt` to `text.README`. Though the renaming of extensions seems unlikely as they affect the interpretation of files, the author could craft the application to accept different extensions for certain files. Furthermore, this technique may lead to higher false positive rates because two completely different files with the same extension would have the same label and result in higher similarity values.

To fully address the problem of renaming, we can use the contents of the files as the labels. We use the MD5 hashes of the files as a representation of their contents and use those as the labels. We again label directories as "directory" again since applying MD5 hashing to them is not applicable. Modifications to the names of files have no effect on the representations because we only look at the contents of files. The problem here is that if an author slightly changes the contents of a file during repackaging, that file will appear completely different from the original file, leading to a false negative.

Another approach is to use the file type from the `file` command as discussed in Section 4.2 to label the nodes. Renaming the files and directories have no impact on this representation because the file types remain the same. This labeling method is more resilient to changes in the contents of files as well, since, for example, changes to the text inside the files will not affect the file type. The author must make specific changes to a file in order for the output of the `file` command to change. For example, for image files the author would have to resize the image, change the image file format, or change the bit representation. For database files, the author would have to change the number of records or change the database version.

A variation of the above approach is to use the first word of the `file` command's file type as the label. This generally leads to using the labels such as "PNG" for PNG image files, "DBase" for database files, and "ASCII" for text files. This variant approach is resistant to changes to the names and contents of files, but like using the file extensions as labels, it may lead to higher false positive rates.

Each of these approaches can encounter different cases that lead to false positives and false negatives. We explore these different labeling methods empirically in Section 4.4.3 and discuss their tradeoffs between false positives and false negatives in Section 4.5.1. We realize that there are other labeling methods available that may improve the representations of directory structures. In this paper, we limit our analyses to these several labeling approaches and show that such representations are generally effective and we leave further improvements as future work.

4.3.2.2 Calculating Differences

After constructing the trees that represent the directory structures of applications, we can compute the tree edit distances among them as explained in Section 4.2. We apply the Zhang-Shasha algorithm to compute these distances, which indicates how different trees are from one another. We then calculate the percent difference between two trees as their edit distance divided by the number of nodes of the bigger tree.

```
Percent difference between tree1 and tree2 =
(tree edit distance) / max(size of tree1, size of tree2) * 100%
```

The smaller the percent difference between two applications, the more similar they are based on their directory structures. After computing all of the percent differences among a dataset of applications, we can sort the results in increasing order so that the pairs of applications with the smallest differences (greatest similarity) appear first. For manual investigation purposes, this process places greater priority onto pairs of applications that have the highest similarity values, and the analyst can set a threshold to examine all applications that are of certain similarity to an application. This process essentially prunes the dataset down to the pairs of applications that the analyst should investigate in detail. As with Juxtapp, we can also use the difference values to perform hierarchical clustering to group similar applications together. We leave clustering as future work, as we only conduct smaller scale experiments in this paper.

One evaluation methodology is to see if the analyst can identify all of the pirated and malicious applications in the sorted list of results given that she will look at a threshold number of pairs of applications. We assume that the analyst investigates the pair of applications with the smallest percent difference first, and then examines the pair that has the next smallest difference, and continues until reaching a threshold number of pairs examined. As an example, let's say we have a sorted list of results in which the pair of applications with the 5th and 31st smallest percent differences are malicious. If the analyst is willing to examine 30 pairs of applications, then she will find the first malicious application but not the second, resulting in a false negative. All of the other applications she investigates are false positives. This methodology may not yield the best results since misbehaving applications may not necessarily appear earlier in the list of results, but repackaged applications do tend to be more similar than completely unrelated applications. As such, pirated applications and also applications that have been repackaged with malware will tend to appear early in the sorted list.

Another investigation methodology is for the analyst to find all applications that are similar to some application, and whether two applications are similar depends on a given threshold. We would need to empirically determine the optimal threshold in this approach. If the threshold for similarity is too high, then we may encounter more false negatives. On the other hand, a threshold that is too low may lead to more false positives. Hence, the threshold is set by the desired precision tradeoff between the "closeness" of applications and the number of applications to examine.

4.3.3 Implementation

The workflow of DStruct consists of the following stages: extracting the APKs, constructing representations of the applications, and calculating differences among the applications. We implemented DStruct in 700 lines of Python.

In the initial step of the process, we extract each APK using the Linux `unzip` command so that we can view the directory structure of the applications. Next, we walk through all of the directories and files and populate the tree along the way as discussed in Section 4.3.2. We create `Node` objects for each directory and file, and each `Node` contains a label and a list of child nodes, which is nonempty only for directories. We label the nodes using the previously discussed methods. For each APK, we would like to create the tree of its directory structure only once and be able to compare it against all other trees. As such, for each constructed tree we serialize and store them using the Python `pickle` module. To access the trees, we simply load them from the stored `pickle` files using the same module and we can proceed with the tree edit distance computations.

To perform the tree edit distance calculations, we use a Python implementation of the Zhang-Shasha algorithm[27]. We construct the trees in the previous step according to the format suggested by this implementation, and we pass in pairs of trees as argument into the `compare` function, which outputs the distances.

Our current implementation DStruct works on a single machine. We use the GNU Parallel program to divide up the tasks to greatly improve performance when dealing with large datasets[31]. Each processor of the machine works on constructing the trees of applications in parallel, and for the pairwise distance calculations, the pairs of applications are split among the different processors as well.

Once the percent difference computations are complete, we simply use the Linux `sort` command to sort the results so that the most similar pairs of applications appear first.

4.4 EVALUATION

In this section we evaluate the efficacy of DStruct. We first introduce the experimental dataset and then we explore case studies in which we use DStruct to detect instances of piracy and known malware on the Android markets.

4.4.1 Experimental Dataset

For our evaluation experiments, we collected Android applications from three different sources. From the official Android Market, we downloaded 30,000 free applications. Additionally, we obtained 28,159 free applications from Anzhi, a third party Chinese market[15].

For experiments dealing with malware, we obtained a dataset of 72 malware from the Contagio malware dump and other sources[16].

4.4.2 Performance

We ran our experiments on Ubuntu Linux 2.6.38 with Intel Xeon CPU (8 cores) and 8GB of RAM. The time to compare trees depends on their sizes, since bigger trees require more operations to calculate their tree edit distances. When we compared one application against all applications from either the official or the Anzhi dataset, the time to complete ranged from 26 minutes to 14 hours. The former time occurred when the application had a tree size of 9 nodes, and the latter time resulted when the application had a tree size of 630 nodes. The average tree size in the official Android Market dataset is 145. This performance is acceptable for the experiments ran for this paper, but for larger scale experiments, we would need a distributed version. Since the pairwise computations are independent, the distributed implementation would be straightforward. We expect that other tree edit distance algorithms can also improve the performance, but at the cost of some accuracy[28][30].

4.4.3 Case Studies

In this section, we examine cases in which DStruct detects similar directory structures among Android applications and allows us to identify instances of piracy and known malware.

4.4.3.1 Piracy

Piracy has become an increasing worry for application developers due to the ease of republishing applications. To upload an application to an Android market, a user simply has to register for that market. Rogue authors often pirate applications from the market, remove the validation and copy protection code, and replace the original developer's revenue generating mechanisms with ones that support the pirate, such as adding advertising libraries.

In an online Guardian article, there were claims that the company Joyworld released a game application called World Wars which is actually a pirated version of The Wars, a game application developed by the company Chillingo[3]. To see if we could detect this case, we paid for and downloaded Chillingo's The Wars as an APK. Using DStruct, we compared this application against the 28,159 applications from the Anzhi dataset using the various node labeling approaches.

Figure 13 shows the list of results when we label the nodes using the first word of the output of the `file` command. The numbers on the left column are the percentage differences between the Chillingo application and the applications listed on the right. We show the 22 pairs with the smallest differences from the sorted results. Using this labeling method, we encounter many false positives. The reason is that the Chillingo application contains hundreds of image files and so do the applications that have small differences from it. Since the labels only include the first word of the `file` data type, DStruct considers most of the image files among the applications the same when they are not. For example, two completely different PNG files are considered the same using this labeling method since they are both labeled as "PNG". Despite encountering false positives, we see three applications marketed by Joyworld, one of them being

```
1.582   com.kcp.quizzflag_18479800_0.apk
1.738   858882713191785739 8f22170d6ed4ae83419c706db0a04cec2.apk
1.887   com.google.book.dunji_52917400_0.apk
2.034   cn.com.nd.momo_13817000_0.apk
2.034   858882805083757682551829376944 1f838115a3ecf783838d7.apk
2.047   com.google.book.douluodalu_50677000_0.apk
2.340   com.SDFighter2_12449900_0.apk
2.355   com.im20.snsvote.activity_32808400_0.apk
2.492   com.mobile.mfklite_78723800_0.apk
2.500   com.concrete.jewelhunt_mcg_71632600_0.apk
2.532   guoguo.ocean.worldwaps_37611100_0.apk
2.532   guoguo.ocean.world_44020000_0.apk
2.660   com.spdg.baby365_64526500_0.apk
2.686   com.baidu.video_29864200_0.apk
2.686   com.qixin.makerwp_61978400_0.apk
2.686   858882623623786939 0ba621e9d9f2800d9d434b9d99442714f.apk
2.690   com.google.book.zhantian_03921500_0.apk
2.848   com.spwebgames.bunny_29410700_0.apk
2.848   org.joyworld.worldwar_24644300_0.apk
2.848   org.joyworld.warser_28437800_0.apk
2.848   org.joyworld.warsers_91689200_0.apk
2.946   com.jim2_11684600_0.apk
```

Figure 13: Sorted list of results when comparing Chillingo's The Wars against the Anzhi dataset. The numbers on the left are the percentage differences between Chillingo's The Wars and the applications listed on the right. We labeled tree nodes using the first word of the `file` output. We show only the smallest 22 results.

```
13.449   858882802773444540 16b7a5500b8c8b88b628e678182df920e.apk
21.044   org.joyworld.worldwar_24644300_0.apk
21.044   org.joyworld.warser_28437800_0.apk
21.044   org.joyworld.warsers_91689200_0.apk
```

Figure 14: Sorted list of results when comparing Chillingo's The Wars against the Anzhi dataset. We labeled tree nodes using the full data type from the `file` output. We show only results with less than 80% difference.

the World Wars game mentioned in the Guardian article. The other two are not mentioned in the article, but upon inspection we found these to be variants of the pirated version. The analyst needs to examine 21 applications before covering these applications. If the analyst, for example, is willing to examine the top 25 applications, then we would consider this a successful case of detecting piracy. However, we can improve our results by using different labels.

Figure 14 shows the list of results when we label the nodes using the full data type outputted by the `file` command. We show only pairs of applications that have less than 80% difference. Only four results appear, meaning this labeling method eliminated all but one of the false positives. Again, we see the three applications marketed by Joyworld. The first application turns out to be a different legitimate version of Chillingo's The Wars, which explains the small 13% difference in directory structures. This is an acceptable false positive because we are expecting different versions of the same application to have similar directory structures. Using this labeling method, the analyst only has to examine 4 applications to detect the cases of piracy.

We repeated the experiment using the names of files as the labels, and we get virtually same result as using the full data type of `file`, except that the percentages are slightly lower.

This indicates that using the file names as labels is a slightly better approach because we do want Chillingo's The Wars to closely match these 4 applications. The last labeling approach we use is the MD5 hashes of files, and again we get the same result, except the percentages range from the 30-50%. These high percentages are not surprising given that any small changes to files that are otherwise identical would cause DStruct to consider them completely different. This is still an effective labeling method given that the analyst still only has to examine 4 applications to identify the pirated ones.

Using the above approaches, we found 3 pirated Joyworld versions of the Chillingo application. These three variants are distinct in application code. The legitimate Chillingo application is unobfuscated, but Joyworld obfuscated the code by renaming the methods and classes in the pirated versions. Additionally, Joyworld added libraries to the applications that were not in the original application. For example, the pirate added advertising and other libraries such as Youmi, Casee, Millennial Media, AdMob, and Wooboo which are used to generate revenue[18, 19, 20, 21, 22]. Finally, the company name of the original application still remains in all pirated versions of the code. Regarding the directory structures, we found that the differences resulted mainly from additions and removals of image files. This confirms the discussion in the Guardian article mentioning that Joyworld modified some of the user interface images. The pirates made no attempt to rename any of the original files. Despite modifications to the directory structure and significant changes to the code, we were still able to detect these pirated versions using DStruct with almost no false positives.

4.4.3.2 Android Malware

With the growth of the Android markets, the incidence of malware has also been on the rise. As of August 2011, it is estimated that as high as 1 million mobile users have been exposed to malware[14]. Google has responded by exercising its remote removal functionality, which allows it to remotely remove malicious applications from affected devices[17]. Google regulates the official Android Market, and we suspect that less regulated, third party markets such as Anzhi will have a higher incidence of malware.

To evaluate whether we can detect known malware in third party markets, we select 5 malicious applications from our malware dataset. These malware samples represent some of the most prolific and well-known malware families[16], including DroidKungFu, DroidKungFu2, GoldDream, zsone, and DroidDream. We compared each of these malicious applications against the Anzhi dataset using similar approaches as in the piracy case study in Section 4.4.3.1. For the node labels, we used the full output of the `file` command and also the file names in separate experiments. Based on our experiments on piracy, these node labels yielded the best results. We leave the other node labels as work for our future experiments.

Table 1 shows that we were able to detect 9 instances of known malware in the Anzhi dataset. We observed that using file names as the node labels generally reduced the number of false positives somewhat as compared to using the `file` output. Among the infected applications that we found on the Anzhi market, the percent differences in directory structure among them and the corresponding applications from the malware sample range from only 0%-10%. Furthermore, all of the malicious applications that we discovered always appeared first in the list of sorted results, indicating that they have the smallest differences from our malware samples. Hence, we did not have to examine many applications before finding malicious ones. In fact, we generally found a malicious application in each case by only looking at the first application in the

Malware	Instances Found
DroidKungFu	5
DroidKungFu2	2
GoldDream	1
zsone	1
DroidDream	0
Total	9

Table 1: Number of instances of each kind of malware found in the Anzhi market dataset.

Application File Name	Repackaged With
com.codingcaveman.solotrial.apk	DroidDream.1
it.medieval.blueftp.apk	DroidDream.2
com.tencent.qq.apk	PJApps
de.schaeuffelhut.android.openvpn.apk	DroidKungFu

Table 2: DStruct is able to detect these original applications that were repackaged with malware when we compared them to our malware dataset.

results. The false positive rates in each case were also low, never exceeding more than 10 false positives. These results show that DStruct is effective in detecting repackaged applications with known malware.

Some of the matching pairs had very minor code variation, but we point out that each of these applications' APKs is distinct on the market since they have different MD5 sums. Typical changes to the discovered malicious applications include modifications of code, file paths, and files. Despite these changes, DStruct effectively detected 9 malicious applications from the Anzhi dataset of 28,159 applications.

We also attempted to find known malware in the official Android Market. Using the same malware samples with an additional one, PJApps, we ran comparisons against the 30,000 applications from the official Android Market. DStruct did not detect any known instances of malware on the official market. This result is unsurprising given that Google has stepped up its efforts to immediately removing malware from the market once it is found and issuing remote removal[23]. However, DStruct did find 4 strong matches to legitimate applications. It turns out that malicious authors had repackaged these original applications with malware to create the corresponding malicious applications in our malware dataset, most likely to entice users into downloading them. Table 2 shows the applications that were repackaged with malware from our dataset. These results show that DStruct is still effective in detecting similar directory structures and cases of repackaged malware.

4.5 DISCUSSION

In this section, we discuss the implications of false positives and false negatives inherent in DStruct and we explore possible approaches to address them. We cover current limitations and suggest future work to improve the efficacy of DStruct. We discuss how using several complementary metrics and techniques together can address false positives and the problem of evasion.

4.5.1 Tradeoffs between False Positives and False Negatives

Depending on how we label the nodes of trees, we encounter a tradeoff between false positives and false negatives in our similarity analysis as demonstrated in our experiments. For example, using the first word of the `file` data type increases the false positive rate as many different files can receive the same label. The similarity comparisons would consider two different PNG files, for example, the same under this labeling approach. On the other hand, this approach is resistant to modifications to the contents and names of the files, so it reduces the

false negative rate. In order to evade detection, the repackager must change the actual data type, such as converting a PNG file to a JPEG file. If we use the full `file` output as the labels, then we reduce the false positive rates as the labels are more descriptive, but this approach is less resistant to certain modifications that can lead to false negatives. For example, the repackager could resize a PNG file, and the comparisons would consider this a different file from the original. Each labeling approach is more resistant to certain modifications than others, at the expense of being less resistant to other changes.

We can imagine using the different types of labeling together. An analyst could first use a labeling method that leads to higher false positives but has greater chances of reducing false negatives. She could investigate those results based on how many applications she is willing to examine, and if she then wants to further reduce the dataset she can rerun on the top results using another labeling approach that leads to lower false positives. The time to run the experiments decreases with each rerun because after running an experiment, the dataset is reduced to only candidates that are more likely to be misbehaving, and any subsequent reruns on the reduced dataset will shrink the dataset further. Another approach would be to use all of the different labeling techniques on the whole dataset. These experiments would require a greater time commitment but can detect misbehaving applications more effectively. In order for repackagers to evade detection, they would have to make specific changes that can evade detection from each labeling method to every file. This places a greater burden on pirates and malicious authors, who have to put more effort into repackaging. Illegitimate authors often would like to keep the appearance of their applications similar to the original ones in order to trick users into downloading them, but this multi-label approach would force them to modify all of the image files at least slightly.

4.5.2 False Positives

Regardless of the chosen labeling approach, false positives can arise due to the fact that different applications will appear similar if their directory structures are similar. For example, we ran DStruct on different versions of an application for several applications and found that their differences in directory structure were all less than 15%. False positives also result from many applications that differ in functionality but were developed by the same author. A common pattern is for a developer to create a framework and use it for a variety of different applications. Another pattern is where different authors create different applications using the same tool for development. These patterns lead to many cases in which the directory structures among different applications appear similar, resulting in false positives. Of course, these patterns do not always hold because authors tend to add, remove, and modify many files in their applications.

Besides applying different labeling approaches, we can reduce the number of false positives using other methods. One observation is that all APK files generally contain certain files and folders, such as `AndroidManifest.xml`, `CERT.RSA`, `res`, `resources.arsc`, and others[32]. Since these files are typically common in all APKs, they contribute to a higher false positive rate, especially when the APK sizes are very small. In light of this, we can exclude these common files and folders that exist in all APKs from our representations of the directory structures. This would increase the percentage differences in virtually all of our results because we would not consider these common files, eliminating more false positives. We would like to implement this improvement as future work.

4.5.3 The Problem of Evasion

Currently, the observed tradeoff between false positives and false negatives is not significant because authors are focusing mainly on modifying code but not the directory structure of applications when repackaging. There are many techniques to detect code reuse as explored in Section 4.1, so authors are attempting to obfuscate the code more to avoid detection, but they often leave the directory structure untouched. As mentioned in Section 4.5.2, when we tested DStruct on different versions of the same application we find high similarity, meaning this approach effectively detects repackaged applications that introduce the same level of modifications to the directory structure that occurs when creating different versions. However, this approach cannot detect cases in which an author develops a malicious application mostly from scratch instead of by repackaging an existing application. Furthermore, if authors understand the details of our approach, they may respond by significantly altering the directory structure when repackaging. For example, the author might add many unused files to or slightly modify the contents of all the files in the application. We can utilize several possible, complementary metrics together to address some of these problems of evasion.

With the tree-based representation of applications, we can employ algorithms to determine if one tree is an isomorphic subtree of another tree[33] as a metric for similarity. Hence, if an author adds many files to the application, the original tree will be a subtree of the new tree, and we will still be able to detect similarity.

As future work, we can explore different representations and comparisons of directory structures to handle the problem of false negatives (and false positives). For example, we can represent directory structures as a hash table in which the keys are the different files and directories and the values are the counts of them. Such a representation is resilient to major reorganizations of the directory structures because the relationships among the files and directories are not considered. For example, moving one file from one directory to another will have no impact on the similarity value. Using this representation, we can apply set comparison techniques to determine similarity. Furthermore, the pairwise set comparisons would run faster than the pairwise tree comparisons because there would be less operations required.

We can also apply more sophisticated techniques to determine the similarity of files. With our current labeling techniques, authors can evade detection by making specific changes to files as previously discussed. When comparing files, we could employ plagiarism[34] and clone detection techniques [11, 24, 25, 26] to determine if they are similar. These approaches are more resilient to modifications to the files and would make it much harder to evade detection.

4.5.4 Similarity Analysis Based on Complementary Approaches

DStruct is a tool that is complementary to Juxtapp, and we can use them together to eliminate false negatives and false positives. In our experiments, DStruct typically encounters much fewer false positives than Juxtapp, though Juxtapp can detect cases of non-repackaged malware[4]. To reduce false negatives, we can apply these two approaches together. Since Juxtapp focuses on application code, whereas DStruct analyzes the directory structure, to completely evade detection one must employ different techniques evade detection from both approaches.

As future work, we should explore ways to use these different similarity analysis tools together. There are many other currently unexplored techniques to determine similarity among

applications, such as examining API calls, the internal structure of files, and the graphical layouts used in applications. In the case of DStruct and Juxtapp, these different approaches can help eliminate each other's false positives, reducing the workload of the analyst. The more techniques we use to detect similarity among applications, the harder it is to evade detection. If we use several detection techniques together, pirates and malicious authors must find sophisticated ways to evade detection from all approaches when repackaging applications. Hence, introducing multiple detection tools like DStruct and Juxtapp help hinder the productivity of illegitimate authors who want to introduce repackaged, misbehaving applications to the markets. Given the widespread and increasing occurrence of piracy and malware on the Android markets, such a multipronged approach to impeding mischievous authors goes a long way to helping the markets stay healthy and sustainable.

5. CONCLUSION

In this paper, we explored the distributed architecture of Juxtapp, which is a scalable infrastructure for detecting code reuse among Android applications. We implemented the system in Hadoop, and we ran it on Amazon EC2. It is capable of fast, incremental additions to the analysis dataset so we are able to update our application repository in an efficient manner. We evaluated the performance of Juxtapp using up to 95,000 applications and find that the system is able to analyze applications rapidly. We introduced a web service that accepts uploads of applications and queries for past analysis results, and it automatically manages the resources required to run distributed Juxtapp on new applications.

In addition, we presented DStruct, a tool for detecting similar directory structures among Android applications, as a complementary approach for similarity analysis. We applied our tool to detect cases of piracy and known malware in a dataset of 58,000 applications from the official Android Market and the Anzhi market. We used DStruct to find 3 instances of piracy in which a pirate removed copyright protection from a paid game and repackaged it with a multitude of advertising libraries that benefit the pirate. We identified 9 instances of known malware with high accuracy and found 4 legitimate applications that malicious authors had used to repackage with malware.

Our experimental results show that Juxtapp is an efficient architecture and that DStruct is a valuable, complementary tool for detecting similarity among Android applications. Given the rise in the popularity of mobile applications and the increased instances of piracy and malware, automatic detection tools such as Juxtapp and DStruct will play an increasing role in improving the security of mobile devices.

ACKNOWLEDGEMENTS

We conducted this research as part of Prof. Dawn Song's group dealing with the security of Android phones. We worked on Juxtapp in collaboration with Steve Hanna, Ling Huang, Edward Wu, Charles Chen, and Dawn Song. We would like to thank Steve Hanna and Ling Huang for providing numerous suggestions and advice for the DStruct approach and manually verifying that the cases of malware mentioned are indeed true positives. We framed parts of this report based on our paper on Juxtapp, and we used the Juxtapp workflow and incremental update figures from the Juxtapp paper. We would like to thank Edward Wu and Charles Chen for

helping with the distributed implementation. We would also like to thank Prof. Vern Paxson for his helpful feedback on DStruct.

REFERENCES

[1] TG Daily. http://www.tgdaily.com/mobility-brief/61070-android-suffers-first-ever-market-share-decline.
[2] Phandroid. http://phandroid.com/2012/04/14/chinese-android-market-share-doubles-in-2011-apple-stays-far-behind/.
[3] Developers express concern over pirated games on android market. http://www.guardian.co.uk/technology/blog/2011/mar/17/android-market-pirated-games-concerns/
[4] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, D. Song. Juxtapp: A scalable system for detecting code reuse among Android applications. To appear in the Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment, 2012.
[5] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Droidmoss: Detecting repackaged smartphone applications in third-party android marketplaces. In Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy, 2012.
[6] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In Proceedings of ACM CCS, 2011.
[7] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In Proceedings of NDSS, 2009.
[8] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In Proceedings of the 4th International Conference on Information Systems Security, 2008.
[9] zynamics bindiff. http://www.zynamics.com/bindiff.html.
[10] X. Hu, T. cker Chiueh, and K. G. Shin. Large-scale malware indexing using function call graphs. In Proceedings ACM CCS, 2009.
[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In Proceedings of ICSE, 2007.
[12] Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps/.
[13] Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps/.
[14] Mobile threat report. https://www.mylookout.com/mobile-threat-report/.
[15] Anzhi android market. http://www.anzhi.com/.
[16] Contagio malware dump. http://contagiodump.blogspot.com/.
[17] Exercising the remote application removal feature. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html.
[18] Youmi advertizing. http://youmi.net.
[19] Casee advertising. http://www.casee.cn.
[20] Millennial media. http://www.millennialmedia.com/.
[21] Admob. http://www.admob.com/.
[22] Wooboo. http://www.wooboo.com.cn/.
[23] Update: Security alert: Droiddream malware found in official android market. http://blog.mylookout.com/2011/03/security-alertmalware-found-in-official-android-market-droiddream/.
[24] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In Proceedings of the 30th international conference on Software engineering, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.
[25] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In Proceeding of the 33rd International Conference on Software Engineering, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.
[26] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering, 32(3), 2006.

[27] Python implementation of Zhang-Shasha algorithm. https://github.com/timtadh/zhang-shasha.
[28] An approximate tree edit distance algorithm. http://code.google.com/p/treeeditdistance/.
[29] Zhang and Shasha tree edit distance algorithm.
http://www.grantjenks.com/wiki/_media/ideas:simple_fast_algorithms_for_the_editing_distance_between_tree_and_related_problems.pdf.
[30] Several implementations of tree comparison algorithms. http://www.inf.unibz.it/~augsten/src/.
[31] GNU Parallel. http://www.gnu.org/software/parallel/.
[32] APK file format. http://en.wikipedia.org/wiki/APK_(file_format).
[33] Isomorphic subgraph problem. http://en.wikipedia.org/wiki/Subgraph_isomorphism_problem.
[34] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In Proceedings of the ACM SIGMOD/PODS Conference.
[35] KilianWeinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In Proceedings of ICML, June 2009.
[36] Dalvik virtual machine. http://www.dalvikvm.com/.
[37] A. Walenstein and A. Lakhotia. The software similarity problem in malware analysis. In Proceedings of Duplication, Redundancy, and Similarity in Software, 2007.
[38] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. Journal of Machine Learning Research, 7, December 2006.
[39] R. O. Duda, P. E. Hart, and D. G. Stork. Pattern Classification. John Wiley and Sons, 2000.
[40] Hadoop. http://hadoop.apache.org/.
[41] Amazon S3. http://aws.amazon.com/s3/.
[42] Amazon EC2. http://aws.amazon.com/ec2/.
[43] Hadoop DistCp. http://hadoop.apache.org/common/docs/current/distcp.html.
[44] SQLite insertion rate. http://www.sqlite.org/faq.html#q19.
[45] Hadoop small files problem. http://www.cloudera.com/blog/2009/02/the-small-files-problem/.
[46] DistributedCache.
http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/filecache/DistributedCache.html
[47] Hadoop Streaming archives.
http://hadoop.apache.org/common/docs/current/streaming.html#Making+Archives+Available+to+Tasks