

Stimulation and Detection of Android Repackaged Malware with Active Learning

Aleieldin Salem

*Technische Universität München
Garching bei München
salem@in.tum.de*

Abstract

Repackaging is a technique that has been increasingly adopted by authors of Android malware. The main problem facing the research community working on devising techniques to detect this breed of malware is the lack of ground truth that pinpoints the malicious segments grafted within benign apps. Without this crucial knowledge, it is difficult to train reliable classifiers able to effectively classify novel, out-of-sample repackaged malware. To circumvent this problem, we argue that reliable classifiers can be trained to detect repackaged malware, if they are allowed to request new, more accurate representations of an app's behavior. This learning technique is referred to as *active learning*.

In this paper, we propose the usage of active learning to train classifiers able to cope with the ambiguous nature of repackaged malware. We implemented an architecture, Aion, that connects the processes of stimulating and detecting repackaged malware using a feedback loop depicting active learning. Our evaluation of a sample implementation of Aion using two malware datasets (*Malgenome* and *Piggybacking*) shows that active learning can outperform conventional detection techniques and, hence, has great potential to detect Android repackaged malware.

1 Introduction

Malware authors targeting Android have recently been adopting a more sophisticated technique to develop and distribute their malicious instances. In essence, they leverage the ease of decompiling, modifying the source code of, and recompiling Android applications (hereafter apps) to repackage legitimate, trusted apps with malicious payloads [29, 13, 17, 26, 35]. This breed of malware is often referred to as either *piggybacked* apps [13, 14] or *repackaged* malware [34, 35]. In this paper, we use the two terms interchangeably to refer to the same type of

Android malware: legitimate apps that have been grafted with malicious payloads.

Repackaged malware can be thought of as an evolution of Trojan horses. The two malware breeds differ in one key aspect viz., the originality of the app's benign functionality. On one hand, Trojan horses usually comprise benign functionalities that have been developed from scratch by the malware author for the sole purpose of presenting the app as a benign one. Furthermore, malware authors of Trojan horses seldom invest adequate amounts of time and effort in developing such fake, benign facade. The resulting benign segment is, therefore, of low quality and limited functionality (e.g., a Sudoku app with five puzzles). On the other hand, repackaged malware comprise pre-existing legitimate apps (e.g., Angry Birds), that have been amalgamated with a newly injected malicious payload (e.g., sending SMS messages to premium rate numbers [9]). The primary threat that repackaged malware poses is undermining user trust in legitimate apps, their developers, and the app distribution infrastructure, which can potentially have devastating effects on the entire Android ecosystem. Unfortunately, malware authors have been increasingly adopting repackaging particularly targeting third-party marketplaces as their distribution platform. In [35], Zhou et al. studied 1260 Android malware instances, and concluded that more than 86% of them were repackaged. In 2015, TrendMicro reported that 77% of the top free 50 apps on the Google Play marketplace had fake versions, with a total of 890,482 fake apps being discovered 51% of which were found to exhibit unwanted and/or malicious behaviors [15]. More recently, Li et al. managed to gather piggybacked versions of around 1,400 legitimate apps [13].

Consequently, the research community has been working towards devising methods to detect Android repackaged malware. To the best of our knowledge, the majority of such methods analyze, and perhaps execute, apps belonging to a dataset (e.g., *Malgenome* [35]), to ex-

tract numerical features used to train a machine learning classifier. The trained classifier is used to classify out-of-sample (hereafter test) apps as malicious and benign [26, 17, 6]. One fundamental problem facing the research community working on detecting repackaged malware is the quality of the ground truth of the currently available datasets. That is to say, a repackaged app is merely labeled as malicious without the knowledge of which paths depict the grafted malicious behaviors, whether there are triggers that control the execution of such behaviors, and how sophisticated those triggers are. Lacking such knowledge hinders training reliable classifiers based on the apps' dynamic behaviors, which is necessary given that malware instances have increasingly been adopting techniques (e.g., obfuscation, dynamic code loading, usage of triggers, etc.), to evade detection by conventional static-based methods [29, 12, 18, 33].

We argue that, in essence, the problem of stimulating, analyzing, and detecting Android repackaged malware is a *search problem*. During the phase of training a classifier, we are in pursuit of the segments in the training apps' source code (e.g., paths in a CFG), that depict or include the injected malicious code. Finding and executing those segments facilitates the extraction of features that resemble their runtime behavior, and enables the classifier to have a collective, accurate view of the malicious behaviors exhibited by Android repackaged malware. Similarly, to reliably classify a test app, we need to execute multiple paths to increase the probability of finding (a representation of) the injected code prior to deciding upon the app's malignance.

Needless to say, the nature and locations of the grafted malicious code in repackaged apps might vary. Given that we lack such ground truth, the training and test processes need to adopt a *trial-and-error* technique. In the training phase, for example, if a feature vector (\hat{x}_i) representing the runtime behavior of a training app (a_i) is misclassified, we assume that such feature vector represents the execution path that does not reflect the app's true nature (i.e., malicious or benign). Thus, the app (a_i) needs to be re-executed to explore another path that yields a feature vector (\hat{x}'_i) that might help the classifier assign the app to its proper class. This process can be iterated until the maximum training accuracy is achieved, which signals the best possible representation of benign and malicious behaviors embedded in the training apps. The number of iterations needed to achieve such maximum accuracy depicts the number of times an app needs to be executed in order to have a comprehensive overview of its behavior. For repackaged malware, it can be interpreted as the number of executions within which the malicious code is likely to execute. The feature vectors corresponding to the different executions can be used to classify the app (e.g., using majority votes), as malicious

or benign.

Within the context of machine learning, the aforementioned trial and error technique is referred to as *active learning*. In this paper, we propose, implement, and evaluate an active learning architecture, called Aion¹ [?], to stimulate, analyze, and detect Android repackaged malware.

Our findings show that—even with primitive stimulation and classification techniques—training a classifier using active learning improves the performance of conventional dynamic-based detection methods on test datasets and can outperform their static-based counterparts. Furthermore, the conducted experiments helped us answer questions about the number of iterations needed to train the best performing classifier, the most suitable kind of features, and the type of the best performing classifier (i.e., Random Trees, K-Nearest Neighbors, Ensemble, etc.). We believe that our experiments provide insights on how to stimulate and detect Android repackaged malware using active learning, and a benchmark against which further enhancements of the architecture (e.g., via using more sophisticated stimulation techniques) can be compared.

The contributions of this paper are:

- We propose a novel architecture and approach to stimulate, analyze, and detect Android repackaged malware using active learning and investigate its applicability and performance.
- We implemented a framework that utilizes the proposed architecture, and made the source code, the data (i.e., gathered during evaluation), and a multitude of interactive figures plotting our results available online [?].
- We evaluate Aion on the recently analyzed and released *Piggybacking* dataset instead of its obsolete *Malgenome* counterpart, and use the former to set a classification benchmark of 72%, similar to Zhou et al.'s 79.6% benchmark on the latter in November 2011.

Organization The rest of the paper is organized as follows. In section 2, we briefly discuss fundamental concepts on which our research is built. In section 3 the methodology adopted during the implementation of Aion, its architecture, and internal structure are introduced. In section 4, we discuss how Aion was evaluated by introducing the datasets we used and the experiments conducted, and discuss the observed results in section 5.

¹Aion is a Greek deity associated with eternity and everlasting time. His orb, depicting repetition, resembles the behavior of our proposed active learning approach, where a feedback loop closes a circle between the processes of stimulation and detection of Android repackaged malware.

Research efforts related to our work are enumerated in section 6. Lastly, section 7 concludes the paper.

2 Background

In this section, we briefly discuss concepts fundamental to follow the remainder of the paper. We discuss our definition of repackaged malware and what it comprises, and introduce the notion of active learning and relate it to our work.

2.1 Repackaged Malware

Android apps can be easily disassembled, reverse engineered, modified, and reassembled [7]. Consequently, repackaging benign apps with malicious code segments is a straightforward process that typically involves the following activities. Firstly, a reverse engineering tool, such as Apktool [1], is used to disassemble an app into a readable, intermediate representation (i.e., Smali). Secondly, a segment of Smali code that delivers some malicious functionality is added and merged with the original Smali code. Lastly, the app is reassembled, re-signed, and uploaded to an app marketplace, possibly with a different name.

The malicious code segments injected into benign apps usually comprise two parts viz., a malicious payload and a trigger (or a hook [13]). The former part depicts the malicious intents of the malware author (e.g., sending SMS messages to premium numbers, deleting the user’s contacts, leaking the current user’s GPS location, etc.). In [9], Fratantonio et al. argue that a malicious payload need not be disconnected from the original app logic; it can distort the actual app functionality, in what they referred to as logic bombs.

The trigger is meant to transfer control to the grafted malicious payload. We can define the trigger as a set of (boolean) conditions that need to hold in order for the malicious payload to execute. For authors of repackaged malware, designing a trigger is a compromise between the likelihood of execution and stealthiness of the injected malicious payload [13]. On one hand, some malware authors may elect to maximize the likelihood of the malicious payload being executed by unconditionally calling/branching to the malicious code (e.g., deployed in a separate method or component) [13]. In this case, the trigger is a *null* condition. On the other hand, some authors may give stealthiness more priority, and develop more complex triggers that activate the grafted malicious functionality depending on specific system properties (e.g., date, time, GPS location, etc.), app/system intents and notifications (e.g., android.intent.action.BOOT_COMPLETED), custom values forwarded to the app via its authors

(e.g., via SMS messages [19]), or app logic (e.g., *if(sum_of_calculation == 50)*). This trend of scheduling the triggering of malicious segments has been found to be adopted by the majority of Android malware [29].

2.2 Active Learning

To the best of our knowledge, the majority of Android malware detection techniques adopt the following process. Firstly, a dataset of benign and malicious apps are gathered. Secondly, the apps in the dataset are analyzed to extract numerical features from each app. That is to say, each app is represented by a vector of features (x_i) and a label (y_i) depicting its nature (i.e., malicious or benign). The extracted feature vectors, often organized in a matrix representation (X), are used to train a machine learning classifier (C) (e.g., a support vector machine). Given a feature vector of an app (x^*) that has not been used during the training phase (i.e., test app), the classifier (C), attempts to predict its corresponding label (y^*) effectively classifying it as malicious or benign. To assess the prediction capability of a classifier, the original labels of the test apps, known as *ground truth*, are compared to those predicted by the classifier. The percentage of correctly predicted labels is known as the classification *accuracy*.

Needless to say, a number of the test apps’ labels are incorrectly predicted (i.e., the corresponding app is misclassified), which negatively affects the classification accuracy. If the classification accuracy is implausible, other classifiers can be probed, the existing features extracted from the apps are further processed (e.g., to eliminate noisy, irrelevant features), a new method of feature extraction is adopted, or a combination of those methods. This setting is referred to as *passive learning* [28]. The passiveness dwells in the inability of the trained classifier to ask for more accurate representations of the misclassified apps in the form of different feature vectors.

In an active learning setting, the classifier is allowed to *query* the sources from whence the feature vectors have been extracted for alternative representations of the same entities [28]. Within the context of Android malware detection, if a feature vector of a test app (x^*) is misclassified, the classifier (C) is allowed to instruct the feature extraction mechanism to generate another feature vectors (x_{new}^*) for the same app. This process can be repeated until either the misclassified app is correctly classified, or the overall classification accuracy of (C) has converged to a maximum value.

3 Design and Implementation

3.1 Overview

The typical malware detection process introduced in the previous section can be grouped into two main sub processes viz., *Data Generation* and *Data Inference*, as seen in figure 1 which depicts our proposed architecture Aion.

The former process commences with the acquisition of a dataset of malicious and benign Android apps. Using a *Simulation* method, the collected apps are analyzed statically and executed within a controlled, virtualized environment to gather information about the apps’ runtime behavior. Within the domain of malware analysis, there are different non-/invasive approaches to stimulating an app [31]. For example, Abraham et al. [4] implemented an approach meant to force the execution of particular branches within an app that contain suspicious API calls, whereas Rasthofer et al. [18] utilized symbolic execution to devise environments (i.e., a set of values) required to execute branches containing specific type of API calls. The monitoring component depicted in the figure is responsible for recording the the stimulated behavior for further analysis. It can be deployed on a remote server, on the environment within which an app is executed, embedded within the app itself (e.g., as logging statements), or a combination of those techniques. Lastly, the monitored behavior may need to be further processed to represent it in a parseable, understandable format. For example, the API calls issued by an app during runtime are usually reorganized in the format of comma-separated strings (i.e., traces).

The stimulated, monitored, and reconstructed representations of the apps’ behaviors are usually raw, and need to be further processed before they are used for training a classifier. Data Inference is concerned with inferring relevant information from the raw data received from its Data Generation counterpart that might facilitate segregating the two classes of apps. The *Projection* component is responsible for processing the raw data and projecting it into a different representation or dimensionality. For example, a trace of API calls can be processed to omit those calls that do not manipulate sensitive system resources (e.g., camera) or extract numerical features (e.g., counts of different API calls). To further remove noise from projected data, the *Inference* component attempts to extract patterns and information that might facilitate the classification of apps. Depending on the format of the projected data, this component can either apply feature selection techniques to rule out irrelevant features, or infer rules that depict characteristics unique to a class of apps. For instance, by studying the API call traces of apps, a rule can be inferred that malicious apps usually use a particular set/sequence of API calls. Lastly,

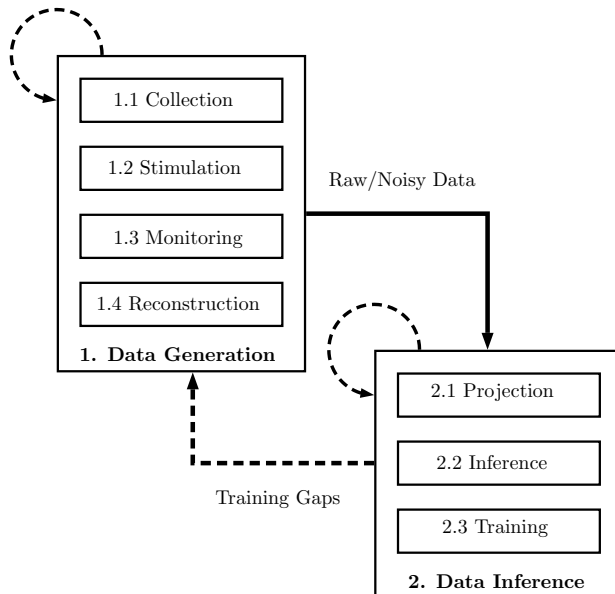


Figure 1: An architecture that uses active learning to stimulate, analyze, and detect Android repackaged malware.

the *Training* component handles training a classifier using the processed raw data, validates the results (e.g., via a test dataset), and reports classification results.

The two dashed arrows looping out of and into the two processes imply that the operations within such processes can be performed multiple times before the process reports its final output. For example, the Data Generation process can continue to add new apps to its repository, stimulate them multiple times, monitor their runtime behaviors, and report an augmented version of the different stimulation sessions. Similarly, the Data Inference process is allowed to consult various combinations of projection, inference, and classification techniques in pursuit of the best classification accuracy.

Lastly, the dashed arrow extending from the Data Inference process to its Data Generation counterpart depicts the active learning aspect of our proposed approach. We illustrate the functionality and significance of such mechanism using the control flow graph (CFG) in figure 2. Consider such CFG as that of a benign app (a_i) that has been injected with malicious segments (colored blocks). Regardless of the technique it employs to target and execute specific branches, the stimulation component should devise test cases that execute the path (P_2) which includes the malicious segments. The feature vectors extracted from the representation of (P_2) (e.g., API call trace) should depict the malicious behavior grafted into the benign app and, in turn, help the trained classifier assign this app to the malicious class. Nevertheless,

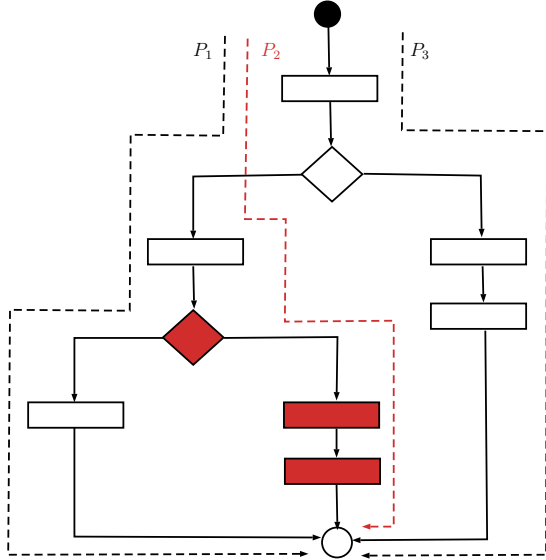


Figure 2: An example of a repackaged malware’s control flow graph (CFG). The colored segment depicts the malicious payload injected into the app, whereas the dashed arrows represent different paths through the CFG.

since the location of the injected malicious payload is unknown, and since the malicious payload is assumed to be smaller than the original benign code, the stimulation component is likely to target other branches/statements within the CFG, effectively executing other paths (i.e., P_1 or P_3). Feature vectors extracted from such paths are expected to represent benign behaviors, leading to the misclassification of the app. The feedback mechanism reports misclassification of (a_i) to the Data Generation process, particularly the stimulation component. Using this information, the latter attempts to target different branches/statements to execute different paths within the app’s CFG, and forward the reconstructed behaviors of such newly-executed paths to the Data Generation process for re-training. In theory, (a_i) will continue to be misclassified until (P_2) is executed, which is expected to help the trained classifier deem the app as malicious. Consequently, the feedback mechanism will continue to instruct the stimulation component to target different code segments until such path is executed.

Given a dataset of training apps, the feedback mechanism is meant to provide the Data Generation module with the best representation of each app that resembles its nature (i.e., malicious or benign). A classifier trained with such representations is expected to possess a comprehensive view of the malicious and benign behaviors exhibited by the apps within the training dataset. However, the large combination of different locations of the injected malicious payloads, their contents and behaviors, and their triggers significantly impedes

the process of executing multiple paths per app. In other words, it can take an unforeseeable amount of time to find the best representation of each app in the training dataset. Furthermore, achieving a perfect training accuracy is unlikely, especially since the behaviors of some apps can mimick those of apps belonging to the other class. That is to say, classification errors are inevitable. In this context, we design the feedback process to terminate once the best possible training accuracy is achieved, which is subject to two constraints. Firstly, the feedback process continues as long as the performance of the classifier increases across two iterations or decreases by a small percentage (e.g., 1%). Secondly, we specify a maximum number of iterations, after which the training process is terminated.

3.2 Implementation

The exact methods, techniques, and algorithms used by different components in the two processes of Data Generation and Data Inference can vary. In other words, the proposed architecture is completely agnostic to specific methods utilized by different components. However, the use of different stimulation approaches might affect the implementation of the Data Inference components and the feedback mechanism. In this paper, we focus on demonstrating the proposed architecture and investigating the applicability of active learning to the problem at hand. Consequently, we adopt an example in which we use basic techniques for stimulation, detection, and feedback mechanisms, and leave the utilization of more sophisticated stimulation/detection methods for future work.

The stimulation technique used in this paper is based on a non-invasive, UI-based tool, called Droidutan [2], which is designed to emulate the user-app interaction. The tool starts the main activity of an app, retrieves its UI elements, chooses a random element out of the retrieved ones, and interacts with it. The interaction hinges on the class of the chosen UI element. For example, if the UI element is a *Button*, Droidutan will tap it, if it is a *TextField*, a random text will be typed into it, and so forth. The tool replicates the same behavior with any activity that may start during the stimulation period. To simulate the occurrence of system notifications or inter-app communication, the tool randomly broadcasts intents declared and registered to by the app in its manifest file.

Interacting with an app generates a runtime behavior that we define in terms of API calls. We use droidmon as our monitoring component to intercept and record a particular set of API calls that interact with system resources such as the device’s camera, the user’s contacts, the package installer, the GPS module, etc. The tool records the intercepted calls along with their arguments and return

results, and writes them to the system log. For each app, we retrieve the API calls from the log and represent the app’s behavior as a series of calls, each having the format `[package.module.class].[method]`.

The APK archives of the apps under test and their corresponding traces retrieved from droidmon’s output are used to extract static and dynamic features, respectively. After surveying the literature, we concluded that static features can be categorized into basic metadata features extracted from the `AndroidManifest.xml` file [21], features related to the permissions required by the app [20, 11], and features related to the API calls found within the app’s code [30]. For each app, we extract static features depicting the aforementioned three categories. We refer to those features as *basic*, *permission*, and *api* features, and we list them in appendix A. The dynamic features extracted from each app’s API calls trace is a feature vector of 37 attributes, each of which depicts the number of times a method belonging to one of the API packages hooked by droidmon, listed in appendix B, has been encountered in the trace.

The static and dynamic feature vectors extracted from the apps’ APK’s and their traces are used to train a majority vote classifier (i.e., Ensemble classifier), using a total of 12 classifiers. The classifiers used to train the Ensemble classifier are K-Nearest Neighbors (KNN) with values of $K = \{10, 25, 50, 100, 250, 500\}$, random forests with values of *Estimators* (E) = $\{10, 25, 50, 75, 100\}$, and a Support Vector Machine (SVM) with the linear kernel.

We assess the performance of different classifiers according to two metrics viz., F1 score and specificity. The first metric is defined as $F1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$ such that $\text{precision} = \frac{TP}{TP+FP}$ and $\text{recall} = \frac{TP}{TP+FN}$. True positives (TP) denote malicious apps correctly classified as malicious, false positives (FP) denote benign apps mistakenly classified as malicious, and false negatives (FN) denote malicious apps mistakenly classified as benign. Thus, the F1 score is meant to assess the ability of a classifier to recognize malicious apps and correctly classify them. We also keep track of the classifier’s performance on benign apps (i.e., whether it classifies them correctly as benign), via $\text{specificity} = \frac{TN}{TN+FP}$ where (TN) stands for true negatives: benign apps correctly classified as benign. We keep track of these two metrics to assess whether a classifier is biased towards one class in favor of the other. For example, if a classifier scores high F1 and low specificity scores, this signals its bias towards classifying the majority of apps as malicious. Detecting a classifier bias can be used to amend the methods adopted to train a classifier (e.g., the type of extracted features).

The apps misclassified by the Ensemble classifier should be, according to our proposed approach, re-

stimulated to execute different segments within their code yielding different paths, different traces of API calls and, consequently, different feature vectors. Given that Droidutan is a random-based testing tool, each run of an app is likely to execute a different path. Consequently, the feedback mechanism in this implementation of Aion comprises merely re-running an app. To ensure that random stimulation results into the execution of different paths, we kept track of the API traces recorded by droidmon for each app. We averaged the number of API calls that change in an apps trace across two consecutive stimulations, and found out that an average of 60 API calls (disregarding their arguments and return values) change with every stimulation.

4 Evaluation

In this section, we evaluate the proposed active learning based approach using two datasets of real world Android malware. The main hypothesis of this paper is that active learning enhances the performance of classifiers trained to effectively detect Android repackaged malware. Consequently, we aspire to answer the following research questions:

Q1: How effective and efficient are the classifiers trained using active learning in comparison to conventional analysis and detection techniques?

Q2: How reliable are the results achieved by such classifiers?

Q3: How well do active learning classifiers generalize on test datasets?

Q4: What is the type of features (e.g., static versus dynamic) and classifiers that yield the best detection accuracies on test datasets?

4.1 Datasets

We used two datasets of malicious and benign Android apps to evaluate our approach. The malicious and benign apps of the first dataset were gathered separately. The malicious apps of such dataset belong to the *Malgenome* dataset [35]. Malgenome originally comprised of more than 1200 malicious apps, almost 86% of which were found to be repackaged malware instances. Prior to being discontinued in 2015, Malgenome was considered the de facto dataset of Android repackaged malware. Fortunately, the repackaged malware belonging to this dataset continue to exist within the *Drebin* dataset [5], from whence we acquired them. To complement the first dataset with benign apps, we randomly selected and downloaded 1800 apps from Google Play store in December 2016. Consequently, we refer to this dataset as *Malgenome+GPlay* in the following sections.

The second dataset, *Piggybacking* [13], is more recent and comprises around 1300 pairs of original apps along with their repackaged versions. The process of gathering the apps, labeling them as malicious and benign, and matching apps to their repackaged counterparts has been carried out between 2014 and 2017. The dataset we used contains 1355 original, benign apps and 1399 repackaged, malicious apps. The reason behind such imbalance is that some original apps have more than one repackaged version.

Lastly, the two datasets were used separately during evaluation. Consequently, some of the experiments we conducted have two variants (i.e., one for each dataset).

4.2 Experiments

To answer the aforementioned research questions, we designed two sets of experiments. The first set of experiments is meant to decide upon the datasets to use in evaluating the proposed active learning approach, and to set a performance benchmark against which the proposed approach is to be compared. The benchmark is set using results achieved from conventional analysis and detection methods that have been previously utilized to detect Android (repackaged) malware. We refer to this set of experiments as preliminary experiments. The second set of experiments evaluates the performance of our proposed active learning approach.

Both experiment sets share a common workflow, seen in figure 3. All experiments commence with randomly splitting a dataset of APK’s into a training dataset (T_R) comprising two thirds of the total number of APK’s and a test dataset (T_E) comprising the remaining one third. The second stage initializes different variables including an initial F1 score of -1 and a variable to keep track of the current iteration’s number (i.e., t). The third stage is responsible for extracting a vector of numerical features (\hat{x}_i) for each APK (a_i) in both the training and test datasets ($T_R \cup T_E$). The method used to extract such numerical features hinges on the type of the experiment run. That is to say, if the experiment is *dynamic*, then the APK’s will be installed on an Android Virtual Device (AVD), run using Droidutan, and monitored using droidmon. However, if the experiment is *static*, the APK’s will be statically analyzed using androguard. Note that for dynamic experiments, droidmon may fail to produce API traces for an unforeseeable number of APK’s due to either (a) the app crashing during runtime, or (b) the app not using any of the API calls monitored by the tool. The third phase concludes with producing feature vectors of numerical features for the APK’s in (T_R) and (T_E). The feature vectors are stored in two datasets ($|X_R| \leq |T_R|$) and ($|X_E| \leq |T_E|$). Stage four uses the feature vectors in (X_R) to train a majority vote classi-

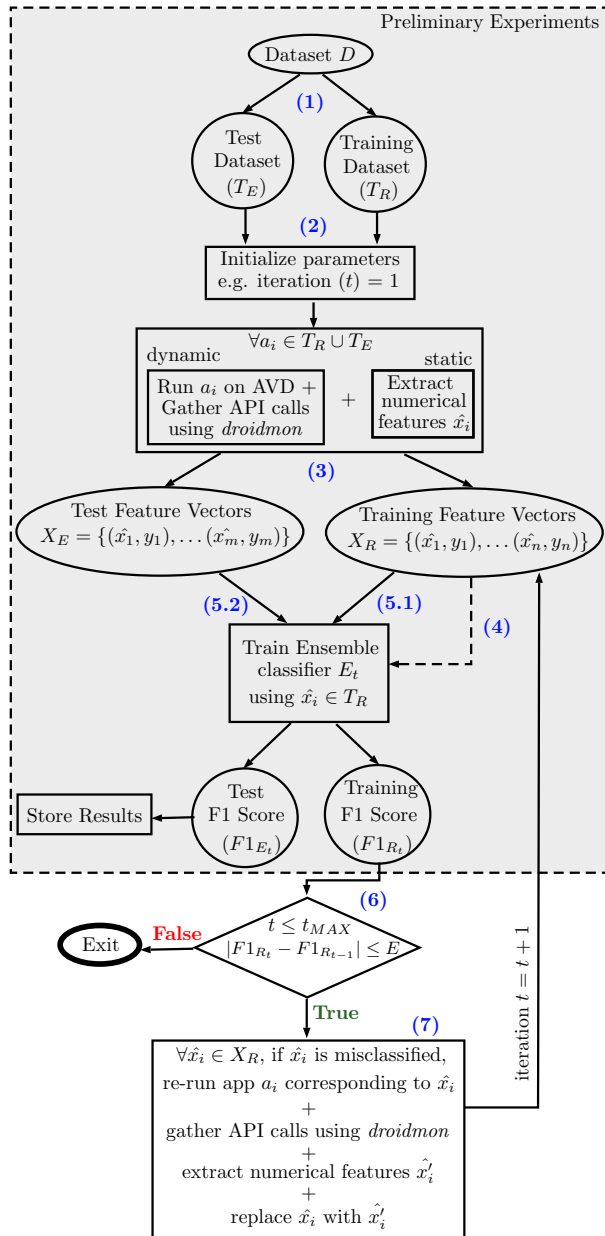


Figure 3: The workflow of our experiments. Ellipses depict inputs and outputs, rectangles depict operations performed on such inputs and outputs, and rhombi depict boolean decisions. The edges between different nodes indicate the directionality of data flow. Lastly, the blue numbers are used as references to different stages of the experiment. Preliminary experiments, whether static or dynamic, conclude after stage (5) and are, hence, highlighted using the dashed gray box.

fier (E_t : Ensemble classifier trained at iteration t), as discussed in section 3.2. In stage five, the trained classifier is validated using the vectors from (X_R) and tested using the vectors from (X_E) yielding corresponding training F1 score ($F1_{R_t}$) and test F1 score ($F1_{E_t}$).

Preliminary experiments conclude after stage five, storing the results into a SQLite database for future study. Active learning experiments compare the training accuracy ($F1_{R_t}$) scored at the current iteration (t) to that scored at the previous iteration ($F1_{R_{t-1}}$). Test accuracies are not used in such comparison to prevent the test dataset (i.e., containing out-of-sample apps), from affecting the training process. This comparison always evaluates to true after the first iteration, especially since the training F1 score is initialized to -1 . For values of ($t > 1$), if the absolute difference between ($F1_{R_t}$) and ($F1_{R_{t-1}}$) is less than or equal to a threshold percentage (E) (default is 1%), the misclassified feature vectors will be re-run in a manner similar to stage three. Intuitively, we continue running the experiment as long as the current training F1 score ($F1_{R_t}$) is increasing. In other words, there is a potential for achieving better training, and perhaps test, scores. Furthermore, in order to accommodate for fluctuations in training accuracies, we allow the current training F1 score ($F1_{R_t}$) to drop for a maximum of (E). This means, however, that the experiment could go on for a prolonged period of time. To avoid this behavior, we set an upper bound (t_{MAX}) for the number of iterations the training phase is allowed to run. In evaluating our approach, we use the values of 10 and 1% for (t_{MAX}) and (E), respectively.

Regardless of the experiment set, a dataset is randomly split into training and test datasets at the beginning of each experiment. The training and test datasets persist throughout the experiment (i.e., through all iterations). Consequently, we run all experiments at least 25 times to increase the likelihood of different segments of the datasets being used as, both, training and test samples. Throughout implementing and evaluating Aion, we generated a multitude of figures, which we could not include in this paper. We host and make available all our (interactive) figures, data, and source code on [?].

4.2.1 Preliminary Static Experiments

In this set of experiments, only the three categories of static features, introduced in section 3.2, are used to classify apps as malicious and benign. That is to say, each app is represented by three feature vectors depicting such categories. Moreover, we combine the three feature vectors to come up with a fourth category of static features that we refer to as *All*.

The scores in figure 4 depict the median F1 scores recorded by different classifiers using all four categories

of static features on the Malgenome+GPlay dataset and its more recent counterpart Piggybacking. To preserve space for more thorough discussion, we opted to move the specificity scores to appendix C. We speculated that solely using static features to classify repackaged malware would yield mediocre classification accuracies, especially since repackaged malware usually poses as benign apps. Nevertheless, as seen in figure 4a, most classifiers perform well on the Malgenome+Play using static features, apart from permission-based features, which apparently are not informative enough to separate malicious and benign apps in this dataset. We noticed, however, that the classifiers comparably underperform in terms of, both, the F1 and specificity scores upon applying the same method to classify apps in the Piggybacking dataset.

We believe there are two reasons behind such noticeable difference. Firstly, the malicious apps in our hybrid Malgenome+GPlay dataset have been gathered between August 2010 and October 2011 [35], whereas their benign counterparts were gathered in December 2016. Given that the Android ecosystem is continuously evolving, any two apps developed 5 years apart might look very different courtesy of newly-introduced technologies, programming interfaces, or even hardware capabilities. In fact, we noticed that there is a large difference between the sizes of APK archives belonging to apps in the Malgenome dataset (average size: 1.25MB) and those belonging to the benign apps we gathered from Google Play (average size: 16.35MB). This difference implies, we believe, the existence of more components (e.g., activities and their classes) in recent Android apps. Static features are built on counts and ratios of those components, the permissions they request, and the API calls they issue. Needless to say, the feature vectors extracted from apps developed in 2011 would look entirely different from those extracted from apps developed in 2015 or 2016, effectively facilitating segregating them in two classes which happen to be malicious and benign.

Secondly, despite being defined as repackaged malware, the malicious apps in Malgenome do not comply with the more recent definition of repackaged/piggybacked apps. In other words, we believe that the 86% of apps comprising repackaged malware in the Malgenome dataset are, in fact, closer to being classic Trojan horses than being repackaged malware. The malicious apps in the Piggybacking dataset, however, comply with the definition of repackaged malware (i.e., standalone benign apps grafted with malicious payloads [13]). To empirically verify this claim, we used a compiler fingerprinting tool, APKiD, to identify the compilers used to compile the apps in both datasets Malgenome+GPlay and Piggybacking along with a dataset of malicious apps called *Drebin* [5]. The distribution of compilers utilized by

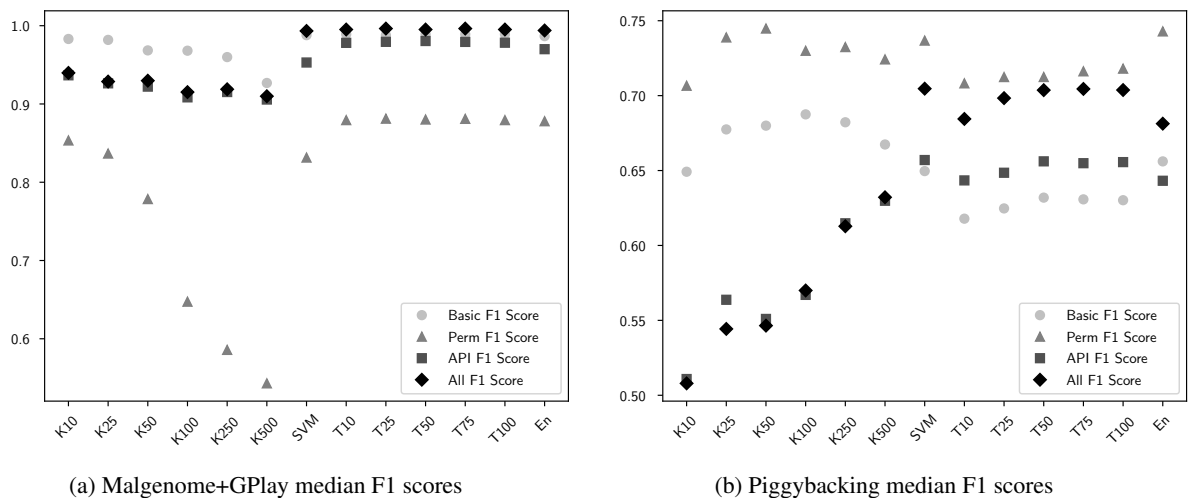


Figure 4: The median (after 25 runs) F1 scores (Y-axis) recorded by 13 classifiers (X-axis) using static features on the test datasets. The specificity scores are plotted in appendix C.

Table 1: Percentages of compilers used to compile APK’s in different malicious/benign datasets as fingerprinted by APKiD. The *dexmerge* compiler is used by IDE’s after *dx* for incremental builds.

Dataset	dx	dexmerge	dexlib 1.X/2.X	Total
Drebin	84%	–	16%	4326
Malgenome	52%	–	48%	1234
Google Play	61%	34%	5%	1882
Piggybacking (malicious)	22%	6%	72%	1399
Piggybacking (original)	61%	22%	17%	1355

apps in such datasets is tabulated in table 1. Using the list of compilers, we rely on the following hypothesis, originally made in [25]: if the compiler used to compile an APK is not the standard Android SDK compiler (*dx*) but rather a compiler used by reverse engineering tools (e.g., Apktool) such as (*dexlib*), then an app is probably repackaged by a party other than the legitimate developers in possession of the app’s original source code.

As seen in table 1, the majority benign apps gathered from Google Play and the original apps in the Piggybacking dataset have been compiled using a compiler usually used within IDE’s. This implies that the developers were likely in possession of the apps’ source code. The same applies to the Drebin dataset which mainly comprises of malicious apps developed from scratch. However, the datasets that presumably comprise repackaged malware indicate a major difference. More than a half of the Malgenome dataset comprises apps that—

according to the definitions in sections 1 and 2—are Trojans, whereas the majority of malicious apps in the Piggybacking dataset comply with the notion of a benign app being grafted with malicious payload and recompiled using non-standard compilers.

Such findings led us to deem the Malgenome dataset as obsolete and inaccurate. In fact, during their analysis of a dataset of 24,650 Android malware samples, Wei et al. also deemed the Malgenome dataset outdated and not representative of the current Android malware landscape [29]. Thus, devising methods to analyze and detect malicious apps in the Malgenome dataset has little benefit to the community. Consequently, we opted to conduct the remainder of our experiments on the Piggybacking dataset, and only consider the static scores recorded on such dataset as our benchmark.

4.2.2 Preliminary Dynamic Experiments

The preliminary dynamic experiments are meant to resemble conventional dynamic approaches to stimulating and detecting Android repackaged malware. That is to say, apps are stimulated once prior to being processed for feature extraction and classification. In essence, such experiments depict the first iteration of our active learning experiments (i.e., without the feedback loop). The features used during such experiments are the dynamic features introduced in section 3.2 along with a combination of the *All* static and dynamic features that we refer to as *hybrid features*.

Figure 5 depicts the median F1 scores achieved by different classifiers on the Piggybacking dataset after 25

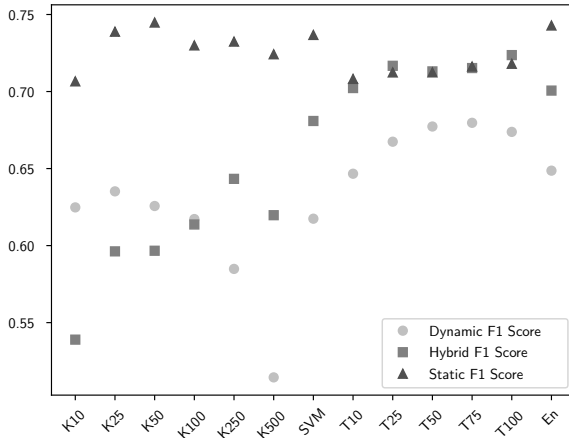


Figure 5: The median (after 25 runs) F1 scores (Y-axis) recorded by 13 classifiers (X-axis) using dynamic and hybrid features versus the best scoring static features (permission-based) on the test datasets of Piggybacking. The specificity scores are plotted in appendix C.

runs using dynamic and hybrid features. As a reference to the performance of static features, we also plot the scores achieved by such classifiers using permission-based static features, which achieved the highest scores on the Piggybacking dataset. In figure 5, dynamic features appear to be incapable of matching the detection ability of static features. One can also speculate that combining dynamic features with static features (i.e., hybrid features) decreases the latter’s detection capability. Nevertheless, dynamic features perform better or slightly worse than other features in correctly classifying benign apps. To conclude, the dynamic preliminary experiments imply that dynamic features extracted after running the apps only once are initially biased towards classifying all apps as benign. We argue that stimulating a repackaged malware once is unlikely to execute the paths under which the grafted malicious payload resides, ultimately leading to the misclassification of such app. In the next set of experiments, we attempt to verify whether re-stimulating the misclassified apps yields different paths that enhance the performance of the detection module and its classifiers.

4.2.3 Active Learning Experiments

Using the Piggybacking dataset, we ran at total of 25 active learning experiments, during which 4 AVD’s were simultaneously used to stimulate apps using Droidutan each for 60 seconds. The misclassified apps were allowed to be re-stimulated until either the difference in the F1 score of the Ensemble classifier drops for more than

a threshold of 1% between two iterations, or an upper bound of re-stimulations is reached. We experimented with such upper bound, and found that with 10 iterations, an experiment takes on average 26 hours to complete. Given that increasing such number did not have a noticeable effect on the achieved scores, and that it substantially increased the time taken to complete one run of the experiment, we adopted an upper bound of 10 iterations.

The scores of the lowest scoring classifier, 500-Nearest Neighbors, the highest scoring classifier, Random forest of 100 trees, and the Ensemble classifier are plotted in figure 6. As seen in the preliminary dynamic experiments, the figures depict the median F1 and specificity scores achieved using dynamic, hybrid, and static permission-based features achieved by the classifiers at every iteration. The static scores are plotted as straight lines, because static experiments did not comprise any iterations. Thus, the scores achieved by a classifier persist across all iterations.

We made the following observations from the plotted figures. Firstly, regardless of the used classifier, dynamic features fail to match or overperform their static counterparts. However, combining static and dynamic features boosts the performance of the latter by around 7%. In other words, dynamic features are incapable of matching static features on their own.

Secondly, the iteration on which the maximum F1 score is achieved depends on the utilized classifier and features type. Furthermore, both the F1 and specificity scores fluctuate across iterations, and do not maintain any specific ascending or descending pattern.

Lastly, regardless of the utilized feature type, some classifiers maintain more stable performance on both the benign and malicious apps than others. For example, for the majority of iterations, the difference between the F1 and specificity scores achieved by the K-Nearest Neighbors classifier in figure 6a is around 20% and 25% for the dynamic and hybrid features, respectively. This behavior indicates bias towards one class in favor of the other. The Ensemble and Random Forest classifiers maintain more balanced performances on malicious and benign apps, which manifests in closer scores across different iterations.

5 Discussion

In this section, we attempt to draw conclusions from the conducted experiments to answer the research questions posed earlier.

With regard to research question **Q1**, we define the effectiveness of an active learning classifier in terms of the F1 and specificity scores it achieves using dynamic and hybrid features on the training datasets in comparison to the scores achieved by the same classifier during

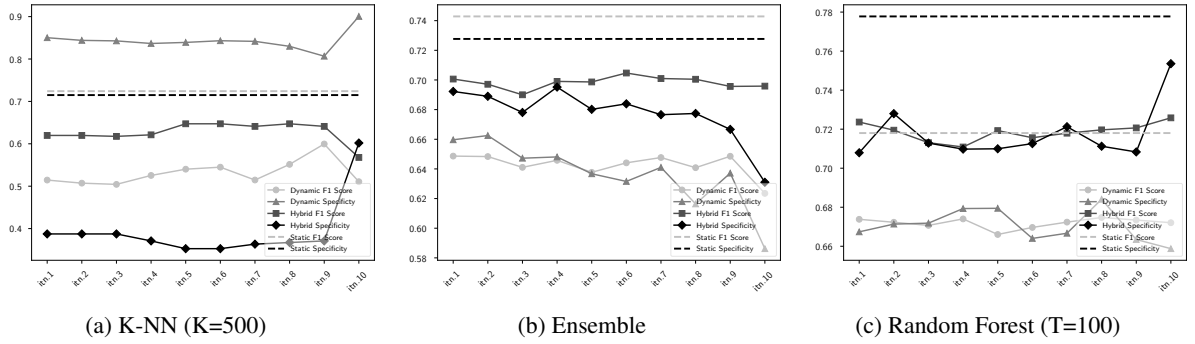


Figure 6: The median (after 25 runs) F1 and specificity scores (Y-axis) recorded by 13 classifiers (X-axis) for each iteration using dynamic and hybrid features versus the best scoring static features (permission-based) on the test datasets of Piggybacking.

the static and dynamic preliminary experiments. We also consider the classifier’s ability to avoid bias towards a specific class as a measure of its effectiveness. We noticed that the aforementioned scores differ from one classifier to another. For example, using dynamic features, the active learning trained 500-Nearest Neighbor classifier scored F1 and specificity scores significantly lower than what it scored during the static preliminary experiments. Furthermore, it was biased towards classifying training apps as benign, which is noticeable via its specificity score in comparison to the F1 score. However, the Ensemble and Random Forest classifiers managed to record F1 and specificity scores that are (a) greater than the scores they recorded during the static and dynamic preliminary experiments, and (b) show low bias given how close both scores are. The scores achieved by all classifiers during different types of experiments can be found on Aion’s *website* [?].

Similarly, the efficiency of active learning classifiers differed from one classifier to another. We define efficiency in terms of the number of iterations it takes a classifier to record its highest F1 and specificity scores on the training dataset. The 500-Nearest neighbors classifier, for instance, achieved its highest scores at the ninth iteration, whereas the Ensemble and Random Forest (with 100 Trees) classifiers recorded their highest scores at the second and third iterations, respectively. We averaged this number across all classifiers, and noticed that it takes around five iterations for an active learning classifier to achieve its maximum training accuracy. As discussed earlier, we consider this number as the number of iterations required by a classifier to have a comprehensive view of the malicious and benign behaviors within the training dataset.

The reliability of the training scores achieved by active learning classifiers, which is the concern of **Q2**, refers to the quality of the training data used to train such clas-

sifiers. In other words, we want to make sure that the classifiers are being trained using traces and features that reflect the malicious and benign behaviors exhibited by current Android repackaged malware and apps, respectively. Otherwise, the classifiers will be trained using inaccurate data that might yield misleading results.

To illustrate this concern, consider the CFG in figure 2. We discussed that, ideally, a repackaged malware (a_i) should be classified as malicious if and only if the malicious path (P_2) was executed. With this research question, we reason about whether the high F1 scores achieved by classifiers such as Random Trees during training honor this condition. In other words, how likely is it that a classifier deems the app (a_i) as benign, given that a path other than (P_2) has executed. Unfortunately, as discussed earlier, we do not possess any ground truth about the paths that reveal the malicious behaviors injected within apps labeled as repackaged malware. Consequently, we can only speculate about the reliability of the results. However, we reasoned about the scenarios under which this behavior could be observed (i.e., a_i is classified as malicious based on benign paths, such as P_1). Firstly, and presumably less likely, the original labels of the apps in the dataset may turn out to be wrong (i.e., a_i is a benign or a harmless app being mistakenly labeled as malicious). Secondly, (P_1) could in fact turn out to be a malicious behavior that we were not aware of and assert as benign. Thirdly, within (a_i), the path (P_1) could be benign; however, it may have been utilized within a malicious context in other apps in the training dataset, which obliges a classifier to deem it as malicious. In the fourth scenario, the path (P_1) could be a benign path that is rarely used by benign apps (i.e., anomalous), which encourages the classifier to deem it as malicious as well. Lastly, there is the possibility of utilizing poor classifiers, features, or learning processes, which yields mediocre, unreliable classification results.

Considering an extreme scenario, assume that all repackaged malware instances in a training dataset have been correctly classified despite the fact that none of their malicious paths has executed. For this condition to hold, we need to assume that all the benign paths deemed malicious by a classifier are anomalous (i.e., seldom encountered in benign apps), which is highly unlikely given the limited number of methods a task (e.g., sending a text message), can be achieved. Moreover, if a classifier mistakenly deems the majority of benign paths as malicious, this behavior should be traceable via the specificity scores which should plummet to minimum values.

To empirically complement the assumptions above, we manually went through the repackaged malware test traces generated by droidmon during the 25th run of the active learning experiments. Around 75% of such test traces were correctly classified using the Random Forest (T=100) classifier trained using active learning. During our analysis, we found evidence that the majority of behaviors based on which the apps were classified as malicious are, in fact, intrinsically malicious. For example, we found that a repackaged version of a simulation gaming app (i.e., *com.happylabs.happymall*), was classified as malicious based on a trace that included retrieving the device’s IMEI, IMSI, and network operator. Furthermore, such trace included the decryption of a URL that pointed to *http://csapi.adfeiwo.com:9999*. Needless to say, such behavior is not expected from a gaming app.

We also examined traces of repackaged malware misclassified as benign apps. We found that the majority of such misclassified traces exhibited benign behaviors. The trained classifier, thus, reasonably assigned them to the benign class. This observation is expected, given that the test apps were stimulated using the primitive tool Droidutan, which is unlikely to execute the malicious behaviors within repackaged malware after one execution. Nevertheless, upon submitting the hashes of the apps corresponding to the examined traces to VirusTotal [3], we noticed that a noticeable number of the malicious apps classified as benign were also deemed as benign by the majority of antiviral software. For example, the repackaged apps *com.sojunsoju.go.launcherex.theme.greenlantern* and *com.gametowin.save_monster.Google* were labeled as potentially unwanted by only two out of more than 50 antiviral software, whereas a repackaged version of *com.gcstudio.DiamondCaveLite* was labeled as benign by all antiviral software on VirusTotal. The latter observation raises the question about the accuracy of the labels in current Android repackaged malware datasets, as discussed earlier.

In absence of the ground truth necessary to verify the reliability of the achieved scores, and based on our previous discussion and observations, our answer to question

Q2 is that we believe that the scores achieved by the active learning classifiers are likely to be reliable.

The active learning process we adopt to train classifiers might lead to overfitting. In other words, the trained classifier would not be able to correctly classify a large number of test apps (i.e., apps not used during the training phase). Research question **Q3** focuses on this issue. Similar to the training phase, we noticed that some classifiers underperform in comparison to conventional training methods, while others can match and outperform them. For example, using hybrid features, Random Forest classifiers trained using active learning manage to outperform conventional training methods using static, dynamic, and hybrid features. Furthermore, it avoids bias towards a certain class by maintaining a good balance between the F1 and specificity scores. Observing the performances of different active learning classifiers, we conclude that Random Forest classifiers are able to achieve the best F1 and specificity scores during, both, training and test phases specifically using hybrid features, which is the answer to **Q4**.

6 Related Work

To the best of our knowledge, there are no research efforts that attempt to apply active learning to the problem of stimulating and detecting Android repackaged malware. However, there are indeed efforts that study repackaged malware/piggybacked apps and attempt to stimulate and detect it. Moreover, we managed to identify research efforts that apply active learning to detecting Android malware in general. Consequently, we divide our related work section into two sub sections that discuss those two dimensions.

6.1 Repackaged Malware Analysis and Detection

There are different efforts that attempt to detect repackaging in Android apps including [23, 10, 34, 8]. Repackaged apps need not necessarily be malicious, however. An app could be repackaged for the sole purpose of injecting advertisement modules or translation purposes. In this paper, we focus on detecting repackaged malware, or piggybacked apps, that conform with Li et al.’s definition: benign apps that has been grafted with malicious payloads that are possible protected with trigger conditions [13].

In [17], Pan et al. developed a program analysis technique that decides whether an app withholds hidden sensitive operations (HSO) (i.e., malicious payloads), effectively deeming it as repackaged malware. Their technique, called *HSOMiner*, gathers static features from the

app’s source code that capture the code segments’ relationship with their surrounding segments, the type of inputs they tend to check, and the operations they usually perform. The extracted features are used to train and validate a SVM that decides whether an app contains a HSO. Needless to say, if an app contains HSO, it probably is malicious. Tian et al. implemented another static approach to detect Android repackaged malware based on *code heterogeneity analysis* [27]. The approach is based on extract features that depict the dependence of a code segments on one another. In essence, injected code segments should exhibit more heterogeneity to the rest of the segments. That is to say, the malicious payloads injected into benign apps should be logically and semantically independent of the original app portions. Similar to our approach, they used the extracted features to train and compare the performance of four different classifiers viz., K-Nearest Neighbors, Support Vector Machine, Decision Tree, and Random Forest. Lastly, Shahriar et al. use a metric called Kullback-Leibler Divergence (KLD) to detect Android repackaged malware [22]. KLD is a metric depicting the difference between two probability distributions. In their paper, Shahriar et al. analyzed the Smali code of benign and malicious apps, from the Malgenome dataset, to build probability distributions of different Smali instructions. Their detection approach is based on the assumption that repackaged malware would have different distributions for different instructions than its benign counterpart.

6.2 Detection with Active Learning

Active learning has been applied to the problem of malware detection on the Windows operating system [16] and within the network intrusion detection [24] domain. In [32], Zhao et al. attempt to apply the same technique to the problem of Android malware detection. Despite adopting a similar architecture that comprises a characteristic (monitoring and extraction) module and a detection module, their work differs from ours in two main aspects. Firstly, Zhao et al. do not consider repackaged malware, rather Android malware in general. Secondly, and more importantly, their definition and utilization of active learning is different from ours. In [32], active learning is defined as a technique to reduce the size of the training samples used to train a SVM classifier that deem an Android app, represented by a behavioral signature (i.e., API call trace), malicious or benign. This reduction is carried out as follows. A pool of pre-generated and labeled behavioral signatures is built as a source for training data. Given an unlabeled, out-of-sample signature, a classifier is trained using the subset of the training pool that guarantees maximum classification accuracy. The main difference between this method and ours

is that the classification accuracy in [32] solely depends on the combination of the training samples, whereas ours depends on the content of such samples, especially since the training vectors usually differ from one iteration to another. Furthermore, this approach to active learning is very likely to yield small training datasets that hinder the trained classifiers from achieving good classification accuracies on test datasets (i.e., *generalization*).

7 Conclusion

The lack of ground truth about the nature and location of malicious payloads injected into benign apps continues to hinder the development of effective methods to stimulate, analyze, and detect Android repackaged malware. We argue that training an effective classifier can be achieved if the classifier is allowed to request better representations of the training apps (i.e., via active learning).

Using a sample implementation of our proposed active learning architecture Aion, we used active learning to stimulate, analyze, and detect Android repackaged malware. In [34], Zhou et al. managed to detect malware in the Malgenome dataset with a best case of 79.6% accuracy [34], highlighting the difficulty of detecting such breed of malware. In this paper, our active learning classifiers managed to achieve test F1 scores of 72% using the more recent dataset Piggybacking [13], which we consider a benchmark for future comparison.

We consider this effort as the first step towards designing better app stimulation, analysis, and detection techniques that focus on Android repackaged malware, which continues to pose a serious threat [29]. Our future plan to enhance Aion has three dimensions. Firstly, we plan on using the number of iterations it takes to train a classifier, and verify whether it transfers to the number of executions needed to effectively classify a test app. Secondly, we plan to use more sophisticated stimulation engines and compare their effect on the training and test scores compared to the random, primitive technique used in this paper. Thirdly, we plan to utilize features and classifiers that capture the semantics of the runtime behaviors exhibited by under test.

References

- [1] Apktool: <https://goo.gl/eaewql>.
- [2] Droidutan: <https://goo.gl/7xsczq>.
- [3] Virustotal: <https://www.virustotal.com/en/search>.
- [4] ABRAHAM, A., ANDRIATSIMANDEFITRA, R., BRUNELAT, A., LALANDE, J.-F., AND VIET TRIEM TONG, V. GroddDroid: a Gorilla for Triggering Malicious Behaviors. In *10th International Conference on Malicious and Unwanted Software (Fajardo, Puerto Rico, 2015)*, IEEE Computer Society.

- [5] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., AND RIECK, K. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS* (2014).
- [6] ARSHAD, S., SHAH, M. A., KHAN, A., AND AHMED, M. Android malware detection & protection: a survey. *International Journal of Advanced Computer Science and Applications* 7 (2016), 463–475.
- [7] BARTEL, A., KLEIN, J., MONPERRUS, M., ALLIX, K., AND LE TRAON, Y. Improving privacy on android smartphones through in-vivo bytecode instrumentation.
- [8] DESHOTELS, L., NOTANI, V., AND LAKHOTIA, A. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014* (2014), ACM, p. 3.
- [9] FRATANONIO, Y., BIANCHI, A., ROBERTSON, W., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 377–396.
- [10] HANNA, S., HUANG, L., WU, E., LI, S., CHEN, C., AND SONG, D. Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2012), Springer, pp. 62–81.
- [11] HUANG, C.-Y., TSAI, Y.-T., AND HSU, C.-H. Performance evaluation on permission-based detection for android malware. In *Advances in Intelligent Systems and Applications-Volume 2*. Springer, 2013, pp. 111–120.
- [12] LI, L., BISSYANDÉ, T. F., PAPADAKIS, M., RASTHOFER, S., BARTEL, A., OCTEAU, D., KLEIN, J., AND LE TRAON, Y. Static analysis of android apps: A systematic literature review. *Information and Software Technology* (2017).
- [13] LI, L., LI, D., BISSYANDÉ, T. F., KLEIN, J., LE TRAON, Y., LO, D., AND CAVALLARO, L. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security* 12, 6 (2017), 1269–1284.
- [14] LI, L., LI, D., BISSYANDE, T. F. D. A., KLEIN, J., CAI, H., LO, D., AND LE TRAON, Y. Automatically locating malicious packages in piggybacked android apps. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems* (2017).
- [15] LUO, S., AND YAN, P. Fake apps: Feigning legitimacy, 2014.
- [16] NISSIM, N., MOSKOVITCH, R., ROKACH, L., AND ELOVICI, Y. Novel active learning methods for enhanced pc malware detection in windows os. *Expert Systems with Applications* 41 (2014), 5843–5857.
- [17] PAN, X., WANG, X., DUAN, Y., WANG, X., AND YIN, H. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps.
- [18] RASTHOFER, S., ARZT, S., TRILLER, S., AND PRADEL, M. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *Proceedings of the 39th International Conference on Software Engineering* (2017), IEEE Press, pp. 300–311.
- [19] RASTHOFER, S., ASRAR, I., HUBER, S., AND BODDEN, E. How current android malware seeks to evade automated code analysis. In *IFIP International Conference on Information Security Theory and Practice* (2015), Springer, pp. 187–202.
- [20] SANZ, B., SANTOS, I., LAORDEN, C., UGARTE-PEDRERO, X., BRINGAS, P. G., AND ÁLVAREZ, G. Puma: Permission usage to detect malware in android. In *International Joint Conference CISIS'12-ICEUTE' 12-SOCO' 12 Special Sessions* (2013), Springer, pp. 289–298.
- [21] SATO, R., CHIBA, D., AND GOTO, S. Detecting android malware by analyzing manifest files. *Proceedings of the Asia-Pacific Advanced Network 36* (2013), 23–31.
- [22] SHAHRIAR, H., AND CLINCY, V. Kullback-leibler divergence based detection of repackaged android malware.
- [23] SHAO, Y., LUO, X., QIAN, C., ZHU, P., AND ZHANG, L. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACM, pp. 56–65.
- [24] STOKES, J. W., PLATT, J. C., KRAVIS, J., AND SHILMAN, M. Aladin: Active learning of anomalies to detect intrusion, 2008.
- [25] STRAZZERE, T. Detecting pirated and malicious android apps with apkid, 2016.
- [26] TAM, K., FEIZOLLAH, A., ANUAR, N. B., SALLEH, R., AND CAVALLARO, L. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 76.
- [27] TIAN, K., YAO, D., RYDER, B. G., AND TAN, G. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Security and Privacy Workshops (SPW), 2016 IEEE* (2016), IEEE, pp. 262–271.
- [28] TONG, S. *Active learning: theory and applications*. Stanford University, 2001.
- [29] WEI, F., LI, Y., ROY, S., OU, X., AND ZHOU, W. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2017), Springer, pp. 252–276.
- [30] WU, D.-J., MAO, C.-H., WEI, T.-E., LEE, H.-M., AND WU, K.-P. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on* (2012), IEEE, pp. 62–69.
- [31] XUE, L., ZHOU, Y., CHEN, T., LUO, X., AND GU, G. Malton: Towards on-device non-invasive mobile malware analysis for art.
- [32] ZHAO, M., ZHANG, T., GE, F., AND YUAN, Z. Robotdroid: A lightweight malware detection framework on smartphones. *Journal of Networks* 7, 4 (2012), 715–722.
- [33] ZHAUNJAROVICH, Y., AHMAD, M., GADYATSKAYA, O., CRISPO, B., AND MASSACCI, F. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (2015), CODASPY '15, ACM, pp. 37–48.
- [34] ZHOU, W., ZHOU, Y., JIANG, X., AND NING, P. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy* (2012), ACM, pp. 317–326.
- [35] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 95–109.

Appendix: Features

A A: Static Features

In this section, we list the categories of static features extracted from the apps' APK's, and utilized in, both, preliminary and active learning experiments

Table 2: The static features used during the preliminary static experiments.

Category	Feature
basic	Min. SDK version
	Max. SDK version
	Total # of activities
	Total # of services
	Total # of broadcast receivers
	Total # of content providers
permission	Total requested permissions
	Android permissions ÷ Total permissions
	Custom permissions ÷ Total permissions
	Dangerous permissions ÷ Total permissions
	Counts of API categories listed in here. (Total: 27)
API	

B B: Dynamic Features

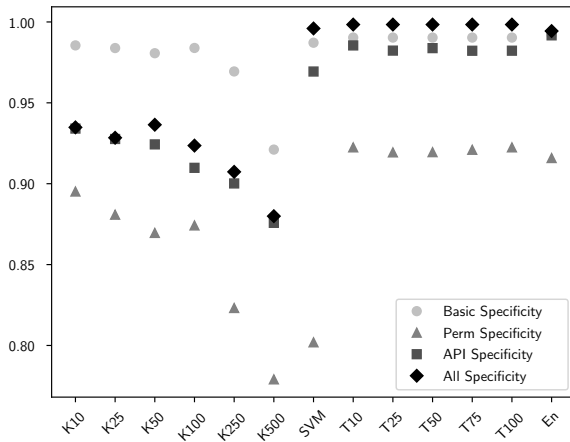
The following table lists the API categories used to extract dynamic features. The total number of API methods hooked is 71 methods, which makes them difficult to be listed in this paper. The exact list of API methods we hook can be found here.

C C: Figures

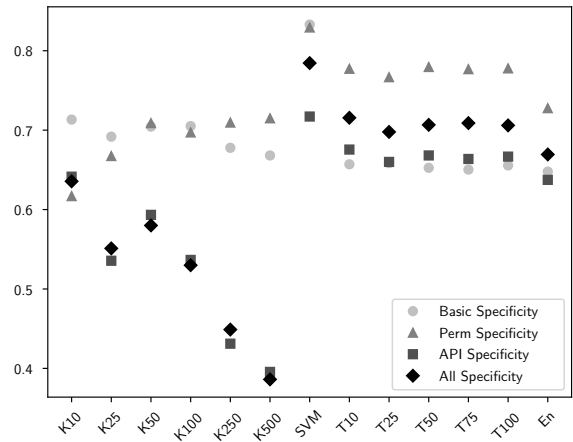
This section contains the plots of specificity scores achieved by various classifiers during the static and dynamic variations of the preliminary experiments.

API Category	Total Hooked
android.accounts.AccountManager	2
android.app.Activity	1
android.app.ActivityManager	1
android.app.ActivityThread	1
android.app.ApplicationPackageManager	2
android.app.ContextImpl	1
android.app.NotificationManager	1
android.app.SharedPreferencesImpl\$EditorImpl	5
android.content.BroadcastReceiver	1
android.content.ContentResolver	4
android.content.ContentValues	1
android.location.Location	2
android.media.AudioRecord	1
android.media.MediaRecorder	1
android.net.ConnectivityManager	1
android.net.wifi.WifiInfo	1
android.os.Debug	1
android.os.Process	1
android.os.SystemProperties	1
android.telephony.SmsManager	2
android.telephony.TelephonyManager	11
android.util.Base64	3
android.system.BaseDexClassLoader	3
android.system.DexClassLoader	3
android.system.DexFile	2
android.system.PathClassLoader	3
java.io.FileInputStream	1
java.io.FileOutputStream	1
java.lang.ProcessBuilder	1
java.lang.Runtime	1
java.lang.reflect.Method	1
java.net.URL	1
javax.crypto.Cipher	1
javax.crypto.Mac	1
javax.crypto.spec.SecretKeySpec	5
libcore.io.IoBridge	1
org.apache.http.impl.client.AbstractHttpClient	1

Table 3: The API categories that comprise the dynamic features used in the preliminary dynamic and active learning experiments.



(a) Malgenome median specificity



(b) Piggybacking median specificity

Figure 7: The median (after 25 runs) specificity scores (Y-axis) recorded by 13 classifiers (X-axis) using static features on the test datasets.

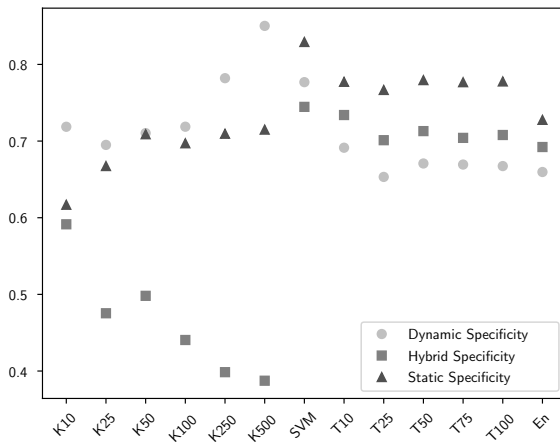


Figure 8: The median (after 25 runs) specificity scores (Y-axis) recorded by 13 classifiers (X-axis) using dynamic and hybrid features versus the best scoring static features (permission-based) on the test datasets of Piggybacking.