MARVIN: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis

Martina Lindorfer[†*], Matthias Neugschwandtner[+*], Christian Platzer[*]
[†]SBA Research, Vienna, Austria
[+]IBM Research, Zurich, Switzerland
[*]International Secure Systems Lab, Vienna University of Technology, Austria
[*]{mlindorfer,mneug,cplatzer}@iseclab.org

***Abstract*—Android dominates the smartphone operating system market and consequently has attracted the attention of malware authors and researchers alike. Despite the considerable number of proposed malware analysis systems, comprehensive and practical malware analysis solutions are scarce and often short-lived. Systems relying on static analysis alone struggle with increasingly popular obfuscation and dynamic code loading techniques, while purely dynamic analysis systems are prone to analysis evasion.**

We present MARVIN, a system that combines static with dynamic analysis and which leverages machine learning techniques to assess the risk associated with unknown Android apps in the form of a malice score. MARVIN performs static and dynamic analysis, both off-device, to represent properties and behavioral aspects of an app through a rich and comprehensive feature set. In our evaluation on the largest Android malware classification data set to date, comprised of over 135,000 Android apps and 15,000 malware samples, MARVIN correctly classifies 98.24% of malicious apps with less than 0.04% false positives. We further estimate the necessary retraining interval to maintain the detection performance and demonstrate the long-term practicality of our approach.

***Keywords*-mobile security; malware analysis; classification**

I. INTRODUCTION

Android is the most popular smartphone operating system today, with a market share of 84.7% [1]. In contrast to Apple's iOS, which restricts users to the applications (apps) available in the iTunes App Store, Android users are not limited to the official Google Play Store. Instead, they can choose from many arbitrary sources, such as third-party application markets, torrents, or direct downloads. Naturally, this liberty causes Android to attract the attention of malware authors, who try to lure users into running malicious code, e.g. by repackaging it with paid or very popular apps. Although there have already been some drive-by download sightings for Android [2], user-based installation is still the most prevalent infection vector.

While more recent studies also found alternative markets hosting up to 5-8% malicious apps [3,4], the official Google Play Store is not free from malware either [5]–[7]. The anonymity of the developer accounts, the low sign-up fee of $25 USD, and its popularity amongst users make the Google Play Store an attractive target, even when considering that developers are banned for life when they are caught uploading malware. As a countermeasure against the growing mobile malware epidemic Google introduced *Bouncer* in February 2012 [8]. By testing apps for anomalous behavior in Bouncer's dynamic analysis environment before listing them in the Play Store, they claim to have reduced malicious app downloads by 40%. Not much is known about the exact functionality of Bouncer, although two independent studies showed that it can be fingerprinted and bypassed easily, just like any analysis environment as long as it is acting as an oracle [9,10]. In November 2012, Google further extended Android's security features by integrating an application verification service in Android 4.2 that is capable of checking apps from any source for malicious functionality. However, an assessment of the effectiveness of this service on a corpus of known malware showed a low detection rate of only 15.32% [11].

Ultimately, when deciding whether or not to install an app, the end-user can consult various information sources:

- Trustworthiness of the app's origin
- App reviews by other users
- Permissions required by the app
- Results from antivirus (AV) scanners
- Results from Google's app verification service

All of these sources have major shortcomings. To begin with, trustworthiness is hard to establish – as stated above – even the Play Store is not safe from malware, and the trust end-users might put into it is likely making it an even more tempting target for malware authors. The problem with app reviews written by users is that most are quite unlikely to notice malicious behavior and their ratings mainly focus on functionality and performance instead of privacy risks [12]. Furthermore, malware authors can use app rank boosting services to increase download numbers and post fake reviews to encourage users to install their malicious apps [13]. The permissions required by an app might actually indicate what the app could do, but this information is too detailed and thus incomprehensible for a majority of users [14]. Finally, several AV companies offer solutions for mobile devices, however, they are restricted by the limited resources and privileges on mobile devices [15,16]. Current devices are not designed to accommodate heavy-weight security solutions including behavior-based malware tracking. One reason is that most Android installations lack the needed root privileges to carry out the necessary operations [17]. Furthermore, a constant runtime scan of running apps puts enormous strain on a device's CPU and therefore reduces battery life drastically.

Consequently, researchers started to leverage machine learning techniques to classify apps based on features learned from known benign and malicious apps. We show in Section II that existing approaches have severe limitations in the their feature extraction as well as their ability to generalize in a real-world setting. We address these shortcomings by extending the large-scale public Android malware analysis sandbox ANDRUBIS [18] to provide users with a risk assessment in the form of a *malice score* that can be efficiently calculated and that is easy to grasp and understand. MARVIN follows the hybrid analysis approach and leverages static and dynamic analysis, both performed off-device (in the cloud), to represent

properties and behavioral aspects of an app through a rich and comprehensive feature set. Using a classifier that is trained on a large set of known malicious (malware) and benign apps (goodware), MARVIN estimates the risk associated with a previously unknown app. By providing a detailed analysis report in addition to the malice score of the classifier our system is both transparent in its assessment and beneficial to both novices and expert users alike. We further evaluate the long-term benefits and practicality of our approach by showing that it can be efficiently retrained and that it maintains its detection accuracy over time.

In summary, our contributions are as follows:

- We introduce MARVIN, a system to automatically evaluate the risk of unknown Android apps through a combination of static and dynamic analysis.
- To provide an appropriate end-user experience, we developed a mobile app that allows users to submit apps to MARVIN and receive malice scores along with a detailed analysis report.
- We evaluate MARVIN on a data set of over 135,000 Android apps, including 15,000 malicious apps, on which it correctly identifies 98.24% of malware samples with less than 0.04% false positives.
- We discuss the most distinguishing features, and features unique to Android apps, that MARVIN uses to classify apps.
- We made our solution available to the public by integrating it in ANDRUBIS, a large-scale app analysis sandbox that accepts submissions via our mobile app[1] and at `https://anubis.iseclab.org`.

II. RELATED WORK

In the domain of Windows malware, machine learning has been extensively used for different purposes in the past. One application of unsupervised learning is finding clusters of samples that exhibit similar behavior based on dynamic analysis [19,20]. In the field of supervised learning Rieck et al. [21] trained classifiers with behavior observed during dynamic analysis to distinguish malware families. ForeCast [22] uses a classification approach to establish a mapping between the static features of malware and a behavioral cluster a certain sample belongs to in order to pre-select interesting samples for dynamic analysis.

Countermeasures Against Android Malware. As one of the main differences compared to countermeasures against Windows malware, related work in the domain of smartphone malware identified the limited processing resources of smartphones and the need to move security defenses off-device and to the cloud. SmartSiren [15] uses collaborative virus detection and detects anomalies in the communication activity collected from a smartphone running its agent. Oberheide et al. [16] also proposed reducing on-device resource consumption and software complexity by sending samples to a server where they can be scanned with a behavioral detection engine. Paranoid Android [23] replicates the execution of an app in the cloud to perform resource intensive detection techniques, such as AV scanning or dynamic taint analysis, to identify exploits. ThinAV [24] modifies the Android Package Installer to consult web-based AV scanners before installing an app. MARVIN can be integrated into this system and provide its malice score in addition to AV scanning results.

Detection Based on Static Features. Peng et al. [25] were the first to propose risk scoring for Android apps to improve the communication of the risk that comes with installing an app to users. They evaluate different probabilistic generative models and based their scoring exclusively on the permissions an app requests. The Android permission system also has been a core element for several other approaches: Kirin [26] ranks security-critical combinations of requested permissions based on a manual assessment. Felt et al. [27] build a simple classification approach based on requested permissions. Another static approach is AppProfiler [28], which alerts end users to privacy-related behavior by matching an app against a knowledge base of over 200 rules of API calls related to critical behavior. The Android Observatory [29] focuses on relationships between apps and provides an interface for users to check whether an app shares its certificate with known malware. Apvrille et al. [30] propose a heuristic engine that statically pre-processes and prioritizes samples to accelerate the detection of new Android malware in the wild. They devised 39 flags for static features that they found to be common in current malware. In contrast, our approach does not rely on heuristics and automatically determines weights of malicious features. MAST [31] considers statically extracted features such as permissions, intent filters, and the presence of native code to perform market-scale triage and to select potentially malicious samples for further analysis. Lastly, Zhu et al. [32] proposed an app recommender system that ranks apps based on their popularity as well as their security risk, but again only consider requested permissions.

Considering our ultimate goal of automatically classifying unknown apps, RiskRanker [33], DroidAPIMiner [34] and Drebin [35] are the most closely related research. RiskRanker detects high- and medium risk apps according to several predetermined features, such as the presence of exploit code, the use of functionality that can cost the user money without her interaction, the dynamic loading of code that is stored encrypted in the app, and the leakage of certain information. DroidAPIMiner and Drebin classify apps based on features learned from a number of benign and malicious apps during static analysis. However, all those approaches lack the ability to analyze code that is obfuscated or loaded dynamically at runtime, a prevalent feature of apps as evidenced by a recent large-scale study [18], unless they are complemented by some form of dynamic analysis, as recently proposed in StaDynA [36]. In contrast, MARVIN does not suffer from these limitations. With a lower false positive rate than DroidAPIMiner and Drebin, MARVIN also classifies more malware correctly (98.24%) than DroidAPIMiner, which reports a maximum detection rate of 97.8%, and Drebin, which identifies only 94% of malware.

The remainder of related work on classifying Android malware based on static features evaluates their approaches on non-representative data sets: The data set of DroidMat [37] only contains 238 malware samples, PUMA [38] on the other hand, in addition to only learning from requested permissions, is trained and tested on only 249 malicious apps. Finally, Sahs et al. [39] also include features from the control flow

[1] Available in the Google Play Store: `https://play.google.com/store/apps/details?id=org.iseclab.andrubis`

graph in addition to permissions. However, they only train on benign samples and use less than 100 malware samples with unknown diversity for verification. Consequently they achieve unusual results: One configuration classifies half of benign applications as malware, while another configuration exhibits a higher false negative than true negative rate.

Detection Based on Dynamic Features. Andromaly [40] is an anomaly detector for Android devices based on on-device monitoring and evaluated different feature selection and machine learning algorithms. However, its evaluation suffers from a lack of available malware samples. STREAM [41] is a framework for evaluating mobile malware classifiers based on the same features as Andromaly with an equally limited testing set of only 50 applications. Additionally, the tested classifiers achieve substantial false positive rates ranging from 14.55% up to 44.36%, rendering them completely impractical.

Crowdroid [42] made a first step towards the use of dynamic analysis results for Android malware detection by performing k-means clustering based on system call invocation counts. DroidRanger [43] combines static and dynamic analysis to perform a large-scale study of malware in several Android app markets. The authors apply a permission-based filtering to improve efficiency and then match apps against behavioral footprints manually extracted from the manifest, API call sequences and the structural layout of an app. Additionally, DroidRanger performs dynamic execution monitoring for apps that match heuristics for the dynamic loading of code in order to detect unknown malware. DroidRanger, however, does not use machine learning techniques to automatically learn discriminative features for classifying malicious apps. Furthermore, contrary to DroidRanger, our approach subjects all apps to dynamic analysis.

Afonso et al. [44] dynamically analyze Android apps to use the number of invocations of API and system calls as coarse-grained features to train various classifiers. Their monitoring approach relies on modifying the app under analysis, which is easily detectable by malware. The only other related approach combining static with dynamic analysis is DroidDolphin [45]. Again, the approach relies on repackaging an app with monitoring code. While the authors observed that the accuracy increased with the size of the training set, DroidDolphin achieves an accuracy of only 86.1% in the best case.

Limitations of Related Work. The majority of related approaches learns from very small data sets, often comprising of only a few hundred apps, and an even more limited and less diverse set of malicious apps. Furthermore, related work failed to investigate the long-term practicality of machine learning techniques for Android malware classification. These practices limit the ability of related approaches to generalize in a real-world setting and their robustness when faced with changes in the Android app landscape. One of the major developments that was observed in a recent study [18] is the increasing importance of dynamic analysis to completely capture an app's characteristic features, in turn, rendering many existing approaches solely relying on static analysis obsolete. Apps can either use reflection and code obfuscation to hinder static analysis [46] or bypass static approaches in general by dynamically loading code at runtime. This code can either be packaged with the app itself, possibly encrypted and even hidden in an innocuous looking image [47], or

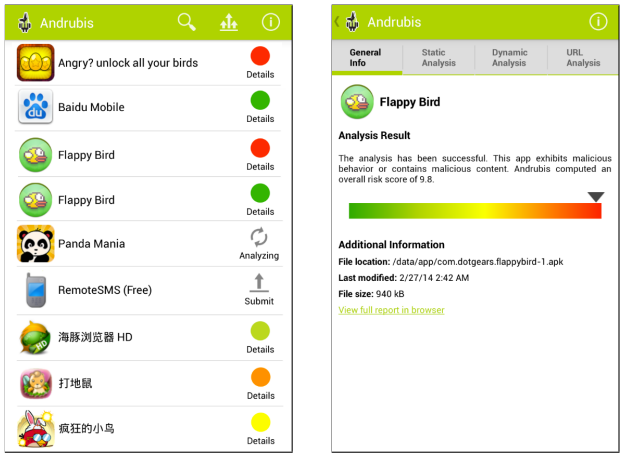


Figure 1: User interface of MARVIN's mobile front end.

downloaded from external sources – making static analysis of the complete app impossible. Such approaches are not only being used by malicious apps, instead, benign applications use dynamic code loading, reflection, and obfuscation alike for application upgrades, statistical A/B testing, premium features (e.g., features purchased in-app), and/or copyright protection.

Most of the existing classification approaches, however, solely rely on static features for classification, thus missing characteristic features in almost 30% of current apps that dynamically load code at runtime [18]. Conversely, approaches relying only on dynamic analysis can be defeated by apps detecting and evading the analysis environment [48,49].

By combining results from static and dynamic analysis MARVIN can automatically identify common features of Android malware and is more accurate and more robust to evasion than prior work. Furthermore, it provides a fine-grained distinction between goodware and malware in the form of a malice score, which is beneficial to both novices and experts.

III. APPROACH

MARVIN learns to distinguish malicious from benign apps based on a set of known malware and goodware. It assigns malice scores to unknown apps in a range from 0 (benign) to 10 (malicious). This addresses the fact that there is a gray area between malware and goodware, e.g. adware, and hence a binary risk assessment would be unsuitable. Furthermore, the scores allow categorizing apps into discrete levels, which makes the results easier to understand for end users [32].

The core element of our risk assessment engine utilizes machine learning techniques that classify apps based on several characteristics. These features are gathered from dynamic and static analysis, network-level behavior and meta-information, like author fingerprints and application lifetime. As a result, MARVIN computes the aforementioned malice scores based on a comprehensive set of distinguishing features that are capable of separating malware from benign apps accurately.

Users can submit apps through a web interface or a dedicated mobile app. The mobile interface is purposely kept simple to attract user attention and to not strain people's patience with overly detailed information that might only be useful to researchers or malware analysts (who can access additional information on a separate screen, if desired). This way, the mobile app is useful for novice and expert users alike.

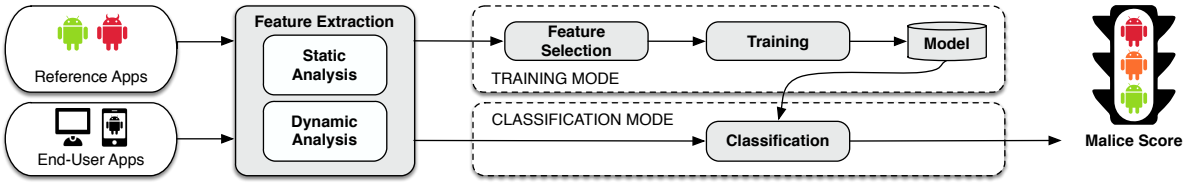


Figure 2: System overview of MARVIN.

Figure 1 shows the app's main screen listing all installed apps, and the analysis overview of a Flappy Birds app repackaged with malware, which was correctly classified as malware with a malice score of 9.8.

In contrast to Google's Bouncer, which scans only apps submitted to the Play Store to approve or reject them, MARVIN analyzes apps from arbitrary sources through a public interface and is independent of a device and its software version. As all operations are performed off-device and independently of a particular installation, MARVIN can serve as a lightweight alternative to AV scanning apps or be integrated into other services. MARVIN can, for example, be utilized by third-party app stores to let users make a more informed decision about which apps to download, or corporate app stores to decide which apps are fit to be distributed further. For the typical Android user, MARVIN acts as an advisor to decide which apps can be safely installed on their device. For expert users and researchers, MARVIN provides an efficient maliciousness rating tool for large-scale evaluations through its public interface.

IV. System Details

MARVIN operates in two different modes (see Figure 2): A *training mode*, where the existing model can be revised and adopted to new features or new strains of malware, and a *classification mode* in which it assesses the risk of an app.

Training Mode. The training mode is needed to learn the model that is later used to evaluate the risk associated with unknown apps. To this end, we provide a training set of known good- and malware to the system. MARVIN first extracts a comprehensive feature set from these apps during static and dynamic analysis. Depending on the configured machine learning algorithm, the features then can be subjected to feature selection to avoid overfitting. Here, the most distinguishing features are determined, which are then used to train the machine learning algorithm and learn the corresponding classification model.

Classification Mode. In classification mode, MARVIN accepts user submissions via a web interface or our own mobile app. Each submitted app is subjected to the same feature extraction as the apps used to train the model. Based on the features exhibited by the app during static and dynamic analysis, and the model created during the training mode, the classifier then assesses the risk and outputs the app's malice score.

The feature extraction step can be skipped, if the submitted app has already been analyzed before. Otherwise, both static and dynamic analysis have to be performed first to gather the necessary features. Therefore, the timeframe to produce a result in this mode can vary from under a second to several minutes if dynamic analysis still has to be performed. However, with a growing repository of cached analysis reports and a daily throughput of over 3,500 new analysis reports, MARVIN is able to instantly assess the risk of a large number of apps – currently amounting to over one million.

A. Feature Extraction

Feature extraction is an essential part of MARVIN. Only a rich and comprehensive set of features that characterizes an app accurately will be sufficient for the classifier to produce meaningful results. Hence, we combine both static and dynamic analysis approaches. For Android apps, static analysis already provides a rich feature set with meta-information about the app, such as its name, requested permissions or registered activities, as well as information about the author from the developer's certificate. Furthermore, static analysis can reveal information that we might not see during dynamic analysis, as the latter might suffer from weak code coverage. Thus, we can statically extract the presence of security-critical API calls and the actually used permissions. Dynamic analysis, on the other hand, shows aspects only observable during runtime, such as dynamically loaded code, even from external sources such as the web, or packed or otherwise obfuscated executables, which are unscrambled during execution. Additionally, capturing the app's network behavior during runtime renders dynamic analysis fundamental for detecting malware reliably.

As a first step in the feature extraction we extract information from an app's manifest. The static analysis also extracts information from the developer certificate used for signing the app and the general structure of the app package. Furthermore, we extract the use of permissions and security-critical API calls from the app's code. Subsequently, we run each app in an emulated analysis environment that runs an instrumented Dalvik virtual machine to record the app's behavior. MARVIN expresses the app's static properties and dynamic behavior as binary features in the format `(S|D)_Category_Name`. For example, a dynamically observed HTTP connection is represented as `D_HTTPGetHost_www.google.com`, while a statically extracted permission request in the manifest is expressed as `S_PermRequired_android.permission.send_sms`. The final outcome of the feature extraction phase is a sparse feature vector $\vec{x}$ of these binary features, with x_i denoting the i-th feature in the vector.

During our large-scale evaluation on a data set of 124,189 unique Android apps, we extracted 496,943 different features (154,939 dynamic analysis features and 342,004 static analysis features). Table I lists the main feature categories and the number of distinct features that we observed for each category. We explain those features in the following paragraphs.

Static Analysis Features. An important source of information for static analysis features is the manifest that has to ship with every app. It contains essential information that the Android OS needs to install and run an app. Among the data contained in a manifest file, we extract the following:

Table I: Categories and numbers of extracted features.

Source	Category	# of Features
static	Class Structure	132,609
static	App Names	93,375
dynamic	File Operations	85,204
static	Certificate Metadata	81,268
dynamic	Network Activity	55,808
dynamic	Manifest Metadata	30,807
static & dynamic	Intent Receivers	10,892
dynamic	Data Leaks	3,662
static & dynamic	Dynamic Code Loading	1,433
static	Used/Required Permissions	1,169
dynamic	Phone Activity	681
static & dynamic	Crypto Operations	35

- The Java package name that uniquely identifies the app in the Google Play Store and many alternative markets.
- The permissions requested by the app. Based on these permissions, the Android OS will grant certain security-critical actions to an app and deny others.
- The intents the app will respond to by the means of a broadcast receiver. Apps can use this feature to be notified e.g. on system boot, the receipt of SMS or incoming calls.
- Publisher IDs for advertisement (ad) libraries. These are used by ad service providers to identify whom to pay the ad view revenue to.

We also add features that indicate whether the Android Application Package (APK) file and the manifest are valid, i.e. they are parseable by standard tools used for analysis, which is not always the case when examining malware samples. Furthermore, we parse the package structure and look for the presence suspicious files, such as native (shared) libraries, native executables and shell scripts embedded in the resources of the packaged app.

Additionally, we statically determine several aspects of the app's code in case they might not be triggered during the dynamic analysis phase. In detail, we extract the following information:

- The used permissions based on the app's API calls.
- The use of the reflection API.
- The use of the cryptographic API.
- The dynamic loading of code, both native code invoked via the Java Native Interface and Dalvik bytecode.

As a large number of apps are submitted to MARVIN without any additional meta-information that would help to identify the author of an app, we rely on the app developer's certificate for authorship information. The certificate used to sign an APK file can be issued by anyone and can be self-signed, but it must be the same for all apps of one author account in the Play Store. Thus, the certificate is also useful for attributing multiple apps to the same malware author. Previous work by Apvrille et al. [30] already suggested that information about the certificates owner/issuer and its validity can be an indicator for malware. Therefore, we extract the fingerprint, serial number, and owner of each certificate, whether it is self-signed and whether its validity period conforms to the release guidelines of the Play Store [50].

Dynamic Analysis Features. As research on x86 malware showed [46], static analysis techniques are prone to evasion by code obfuscation techniques. Furthermore, features should inherently represent the malicious behavior to be detected to prevent attackers from evading the learning method, e.g. with mimicry attacks [51]. Thus, any static analysis is ideally complemented with dynamic analysis that captures the harmful behavior inherent to malicious apps.

In order to obtain the dynamic analysis features, we extended the automated and publicly available dynamic analysis sandbox ANDRUBIS that we proposed in previous work [18]. ANDRUBIS performs monitoring at the Java-level through a modified Dalvik VM as well as at the system-level through virtual machine introspection (VMI) in the emulator. Additionally, it employs various stimulation techniques to trigger program behavior and increase code coverage. We analyze each app for four minutes. This timeframe yielded the best trade-off between the use of our analysis resources and observed features in previous experiments. Furthermore, this timeframe ensures that the user receives his risk assessment in a reasonable amount of time, even for apps that have not been analyzed in the past. During analysis, we monitor the following events:

- *File operations.* Each file operation is represented as a combined feature of the type (read/write) and the file name.
- *Network operations.* Depending on the protocol level, network operations offer various types of information. Starting at the IP level, we represent destination host and port as distinct features. In the case of SMTP, FTP, DNS, HTTP, and IRC communication we also extract additional higher level features. For FTP these are username and password of a conversation, for IRC they are username, nickname, password and channel, and for SMTP we extract the sender's address and the message subject. For DNS requests we extract the queried domain names as well as responses, as unsuccessful DNS resolutions (`NXDOMAIN`) can indicate malicious apps using domain generation algorithms [52]. For HTTP, we create a combined feature of the method and the request. For the request, we remove the request parameters as they showed to be extremely noisy in preliminary experiments.
- *Phone events.* Both outgoing phone calls as well as sent SMS are represented by the corresponding phone number.
- *Data leaks.* Observed data leaks are represented depending on the data sink used. For leaks via SMS we record the phone number, network leaks are expressed by host and port, and leaks to the file system by the file name.
- *Dynamically loaded code.* Code loaded at runtime is represented by the type of code (either native code or a DEX class) and the file name, respectively the class name.
- *Dynamically registered broadcast receivers.* Broadcast receivers are represented by the intent they are registered for.

To reduce the dimensionality of the feature vector and to avoid overfitting, we keep the features as generic as possible by replacing app or runtime specific identifiers such as process IDs, file descriptors and the package name of the app under analysis with tokens.

B. CHOOSING A CLASSIFIER

The classifier is a core component of MARVIN, as its accuracy will immediately be reflected in the malice score. When choosing a classifier, we are bound to the requirements of our domain:

- *High-dimensional feature space.* The number of features in our evaluation data set exceeds 490,000.

- *Sparse data.* The apps in our data set only exhibit a small subset of the possible features.
- *Performance.* Both the training and classification process should take a limited amount of time, to enable short retraining intervals and provide the end-user with an as-instant-as possible risk assessment.
- *Scoring.* Since we want to address the gray area between malware and goodware, a classifier providing only a binary assessment, e.g., a decision tree, is unsuitable for our purposes.

Given these characteristics, we explore two machine learning approaches: a linear classifier and a Support Vector Machine.

Linear Classifier. Given a feature vector $\vec{x}$, a linear classifier computes the scalar product with a weight vector $\vec{w}$: $y = \sum_i x_i w_i$. The outcome, y, is the margin of the classification. In essence, the weight vector $\vec{w}$ can be visualized as a hyperplane that splits the feature space into two sections, representing the classes the classifier can distinguish. While the sign of the margin specifies on which side of the hyperplane $\vec{x}$ is, its absolute value $|y|$ can be interpreted as the confidence in this classification, with larger values corresponding to more confident predictions.

When operating on a binary feature space as in our case, linear classification speed directly scales with how sparse a given feature vector is, because the computational effort of computing the scalar product $\vec{w} \cdot \vec{x}$ is proportional to the number of features expressed by a given sample.

Prior to actual classifying, the classifier's weights need to be determined in a training process. For training we experimented with both, L_1- and L_2-regularized logistic regression. L_1 regularization tries to minimize the sum of the absolute values of the feature weights to be small and tends to assign zero weight for a large majority of the features. In contrast, L_2 regularization optimizes the sum of the squares of the weights to be small and tends to assign non-zero weights to most features. As suggested by Andrew Ng [53] L_1 regularization proved superior to L_2 regularization when dealing with many irrelevant features, while logistic regression with L_2 regularization is extremely sensitive to the presence of irrelevant features. We show in our evaluation in Section V that both methods lead to similar results during classification, while the L_2 classifier performs noticeable better on previously unseen malware samples with outdated training data after several months. This suggests that the implicit feature selection observed with the L_1 classifier is only appropriate when using very short retraining intervals, which is why we chose the L_2 classifier in the current deployment of MARVIN.

Support Vector Machine (SVM). In principle, an SVM works the same way as a linear classifier, with a hyperplane splitting the feature space into two sections. However, it does address one problem of linear classifiers: as the name already suggests, the latter classifies samples only accurately if the problem is linearly separable. To overcome this limitation, SVMs use the "kernel trick", implicitly mapping the input into an even higher-dimensional space, where the problem is more easily separable.

The kernel of an SVM can be seen as the similarity measure. For a pure linear classification it would be $K(x_i, k_j) = x_i \cdot x_j$ (the dot product). Since we want a non-linear classifier we use the standard Gaussian radial basis function (RBF) kernel $K(x_i, k_j) = exp(\frac{-|x_i - x_j|^2}{\gamma})$. For training, an SVM requires the cost factor C, which controls the trade-off between margin and erroneous classification. To determine C and the RBF-specific parameter γ, we perform cross-validation.

As further detailed in Section V, pure linear classification performed at least as accurate as the SVM and can be trained significantly faster. Thus, MARVIN uses a purely linear classifier.

Prediction Probabilities as Scores. In the common case, the final result of a binary linear classifier is given by the sign of the margin: it either attributes the input to one or the other class. However, in the case of a malware/goodware distinction, a more precise differentiation is desirable. For instance, adware might express some features that are characteristic of malware, yet adware itself is not necessarily malicious. To address this problem, we do not want the classifier to merely predict classes, but also output the confidence or probability that an app belongs to a specific class.

To calculate this probability, we use the standard method for probability estimations, an exponential function of the margin: $\frac{1}{1+e^{-y}}$. The higher the amount of malicious features an app under classification exhibits, the further away it is from the boundary of the separating hyperplane and the higher this probability will be. For display purposes we scale this probability to the interval $[0, 10]$ as the malice score for an app. Additionally, we can display the malicious features that contributed most to an app's malice score to let users make a more informed decision whether to trust an app or not.

C. FEATURE SELECTION

The features detailed in Section IV-A include a large number of features that we extract from the static and dynamic behavior of Android apps. However, this does not necessarily mean that they are useful for classification. Some features are available throughout (almost) all apps in our data set while other features are random or simply vary from one app to another. In order to reduce the dimensionality of our feature vector and to use only the most discriminative features, we experimented with feature selection using the F-score (Fisher score) [54]:

$$F(i) \equiv \frac{(\bar{x}_i^{(+)} - \bar{x}_i)^2 + (\bar{x}_i^{(-)} - \bar{x}_i)^2}{\frac{1}{n_+ - 1} \sum_{k=1}^{n_+} (x_{k,i}^{(+)} - \bar{x}_i^{(+)})^2 + \frac{1}{n_- - 1} \sum_{k=1}^{n_-} (x_{k,i}^{(-)} - \bar{x}_i^{(-)})^2}$$

The F-score is calculated on feature vectors $\vec{x}_k,\ k = 1 \ \ldots \ m$, with n_+ and n_- being the numbers of positive/negative samples and $\bar{x}_i, \bar{x}_i^+, \bar{x}_i^-$ being the average of the i-th feature of the whole, positive and negative sets. The higher the F-score for a certain feature i, the more discriminative and important is this feature to the overall classification accuracy.

For the linear classifier, feature selection is implicitly performed depending on the regularization algorithm and we trained both linear classifiers with L_1- and L_2-regularized logistic regression on the same feature set. For the SVM classifier, we performed an additional feature selection step. It iteratively runs its parameter selection with $I, \lfloor I/2 \rfloor, \lfloor (I/2)/2 \rfloor, \ldots, 1$ features ranked by their respective F-score and with I being the total number of features. The final result of this procedure is the set of features that produced the

highest accuracy on the sample set. We used a set of known malware samples and benign apps from the Google Play Store to determine the subset of discriminative features and their corresponding F-score to achieve the best classification results using an SVM with an RBF kernel. We did not include our whole set of labeled data in the feature selection process to avoid overfitting the training data.

V. Evaluation

In this section we provide a detailed evaluation of MARVIN. We present our data sets, our training procedure, and how MARVIN fares when assessing known malicious and benign apps. Furthermore, we evaluate how MARVIN performs on apps from unknown origins, and how well it maintains its classification performance over time. Finally, we investigate the most decisive features when distinguishing benign from malicious apps.

A. Data Set Selection

We analyzed a total of 124,189 Android apps that MARVIN analyzed between June and October 2012 for the original data set used for training and testing. We further analyzed an additional 11,634 apps MARVIN analyzed from January 2013 to May 2014 to evaluate its retraining effectiveness. Table II lists the distribution of apps across all data sets.

Ground Truth. The ground truth and input for our feature selection and training are two labeled data sets: One set is comprised of known benign apps and one set consists of known malicious apps. We collected the benign apps from the Google Play Store and scanned them with VirusTotal [55]. We did not label apps as goodware if they triggered a response from any of the 43 AV scanners used by VirusTotal.

We retrieved the malware data set from VirusTotal and selected samples at random from 30 variants of the 16 most widely distributed malware families according to the F-Secure mobile malware report for the second quarter of 2012 [56]. In order to diversify our malware collection, we extended this data set with 1,894 malware samples belonging to various families that were first seen by VirusTotal in September 2012 and that matched more than 10 AV signatures. Additionally, we include the Android Malware Genome Project [57] and the Contagio malware dump [58] as known malware corpora.

We labeled 78% of the apps in our data set as either goodware (around 68%) or malware (around 10%). The remaining 22% of apps were not labeled. We received them from unknown or untrusted sources mainly through anonymous submissions to the web interface of MARVIN, sharing with other researchers, or torrents and direct downloads from one-click hosting sites. Unlabeled apps also include samples collected from the Google Play Store that were detected by one or more AV scanners. We also did not label samples that we retrieved from VirusTotal that matched below 10 AV signatures or matched signatures for grayware such as adware, spyware and riskware.

Sample Activity. We summarize the number of apps for each label as well as the number of features extracted statically and dynamically in Table III. On average, an app expresses 40 features, with two thirds extracted statically from the app and one third extracted during dynamic analysis. The six features that were always extracted from an APK file are the app's name and features from the certificate. For 0.61% of all apps, the static analysis failed to extract any further static analysis features and only dynamic analysis features remained. The number of dynamic analysis features depends on the activity of a apps in the sandbox and is zero for 5% of our apps that did not exhibit any behavior during the analysis timeframe, but which were nevertheless classified based on static analysis features alone. Apps with both, no static analysis results and no behavior in the sandbox, amounted to only 0.63% of our data set.

Table II: Separation of data set in training and test sets.

Data Set	Total	Malware		Goodware		Labeled
Feature Selection	9,180	4,580	49.89%	4,600	50.11%	✓
Training Set	66,891	7,406	11.07%	59,485	88.93%	✓
Test Set	28,670	3,175	11.07%	25,495	88.93%	✓
Genome Project	1,152	1,152	100%	0	0%	✓
Unknown Set	27,476	-	-	-	-	-
Total	124,189	11,733	9.45%	84,980	68.43%	
Malware Retraining	1,134	1,134	100%	-	0%	✓
Mixed Retraining	10,500	2,874	27.37%	7,626	72.63%	✓
Total	135,823	15,741	11.59%	92,606	68.18%	

Table III: Min,max,mean number of features for all classes.

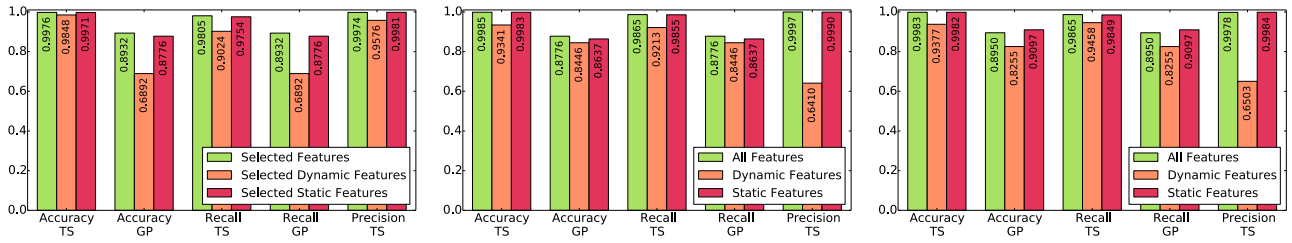
Class	# of Apps	All Features			Static Features			Dynamic Features		
		min	max	mean	min	max	mean	min	max	mean
Malware	11,733	7	196	37.33	6	186	24.04	0	88	13.28
Goodware	84,980	6	1,023	34.58	6	1,019	25.60	0	488	8.9
Unknown	27,476	6	1,529	40.72	6	1,510	29.82	0	321	10.90
Total	124,189	6	1,529	36.20	6	1,510	26.39	0	488	9.81

Training and Test Set. We randomly split the set of labeled apps in a training set (70%), and a test set (30%). Both sets contain 11.07% malicious and 88.93% benign apps, reflecting the proportion of malware to goodware in submissions to ANDRUBIS in November 2012. The malware set includes samples from the most prevalent families according to F-Secure, Contagio, and the samples we collected from VirusTotal in September 2012. We retained samples from the Genome Project as an independent test set to verify MARVIN's classification accuracy on previously unseen malware. We also retained a set of labeled apps for feature and parameter selection of the SVM.

B. Classification Results

As we have detailed in Section IV-B, we experimented with three different classifiers: an SVM with RBF kernel and two linear classifiers with L_1- and L_2-regularized logistic regression. We evaluated both the overall classifier performance in terms of accuracy, and the time necessary to train the classifier. As we show in this section, all classifiers produced comparable classification results although with vast differences in training and testing times. The SVM with the RBF kernel and preselected features took on average 16.5 minutes to train and 27 seconds to classify the full test set (on average 0.95 ms per sample). The linear classifier with L_1 and L_2 regularization on all features took only two to three seconds for training and under a second for classifying the test set (on average 0.02 ms per sample).

All classifiers output a prediction whether an app is good- or malware and the probability with which the app belongs to either of those classes. In the following, we provide a discussion of the classification results on the test and unlabeled set as well as the samples from the Genome Project.



(a) SVM with RBF kernel. **(b)** Linear L_1-regularized. **(c)** Linear L_2-regularized.

Figure 3: Performance in terms of accuracy, precision and recall on the test set (TS) and the Genome Project (GP). Note that the precision on the Genome Project is always 1.0 because it contains only malware, hence we omitted it from the graphs.

Table IV: Sources of apps with classification results and detection rates by VirusTotal's AV scanners for unlabeled apps.

Data Set Source	Total # of Apps	# of Apps in Training/Test Set	# of Apps in Unlabeled Set	# of AV Detections >5	>10	>20	Malicious according to MARVIN SVM	Linear L_1	Linear L_2
Google Play Store	90,951	85,008	5,943	201	105	3	180	24	101
Torrents	4,241	132	4,109	10	9	1	108	16	24
Direct Downloads	1,565	16	1,549	4	3	0	22	5	5
Sample Sharing	4,716	729	3,987	166	140	71	567	331	382
VirusTotal	10,949	9,950	999	913	783	188	767	759	794
Contagio	324	198	126	126	126	86	112	112	114
Genome Project	1,152	0	1,152	1,150	1,134	454	1,029	1,011	1,031
User Submissions	12,863	0	12,863	3,865	3,104	122	4,739	4,388	4,330
Total Distinct Samples	124,189	95,561	28,628	5,955	4,930	698	7,102	6,244	6,365

Classification of the Labeled Test Set. All three classifiers predict the correct class for the apps in the test set with extremely high accuracy, as illustrated in Figure 3: 99.76%, 99.85%, and 99.83% for the SVM and RBF, the L_1-regularized, and the L_2-regularized linear classifier respectively. The SVM misclassifies 62 malware and eight goodware apps (1.9528% false negatives, 0.0314% false positives), the L_1-regularized linear classifier misclassifies 43 malware and only one goodware apps (1.3543% false negatives, 0.0039% false positives) and the L_2-regularized linear classifier misclassifies 45 malware and seven goodware apps (1.3543% false negatives, 0.0275% false positives).

Taking the base rate of malicious to benign applications in the test set into account, we further calculate the Bayesian detection rate, i.e. the probability that an app classified as malicious by MARVIN indeed is malware. Here, the SVM, the L_1-regularized and the L_2-regularized linear classifier achieve detection rates of 97.98%, 99.74%, and 98.24% respectively.

Another important metric for the practicality of MARVIN is the number of false alarms raised. Considering the false positive rate in relationship to the average number of installed applications on a user's device (around 90 according to a recent study [59]), MARVIN raises false alarms for 0.004 apps with the best configuration (L_1-regularized linear classifier) and 0.025 apps with the worst configuration (SVM). If MARVIN were used to analyze a whole app store, such as Google Play with currently almost 1,500,000 apps [60], MARVIN would raise false alarms for 58.5 apps in the best case with the L_1-regularized linear classifier and 471 in the worst case with the SVM.

Manual examination of the false positives reveals that all misclassified goodware apps raise red flags by requesting between ten and 32 permissions including the permissions to send and receive SMS, start a service on startup, install packages, remount the file system, or modify the APN settings that control the cellular data configuration. One misclassified app is a mobile security app that also includes embedded executables and dynamically loads native code in addition to requesting dangerous permissions. The majority of misclassified malware apps receive very low scores due to their inactivity during dynamic analysis or a limited amount of static analysis features.

Classification of the Genome Project. We excluded all samples from the Genome Project from our training and test set to evaluate the accuracy of our classification on previously unseen malware. This set consists of around 1,200 malware samples from 49 different families and includes the majority of prevalent malware families from August 2010 to October 2011 [57]. Despite the age of this data set, MARVIN classifies close to 90% of the samples correctly. The majority of misclassified samples belongs to only three families: *DroidDreamLight*, *jSMSHider*, and *GoldDream*.

Static vs. Dynamic Features. As illustrated in Figure 3, classifying apps using a combination of static and dynamic analysis features yields the best results, while classification based on static analysis features alone outperforms classification based on only dynamic analysis features. However, dynamic analysis features are the only distinctive features in a several cases and are indispensable for classifying apps dynamically loading code at runtime and describing an app's network behavior. MARVIN could benefit from incorporating more dynamic analysis features, such as system calls, to make the dynamic analysis more decisive, which we leave for future work (see Section VII). Furthermore, in ongoing work we explore the benefit of a more comprehensive GUI stimulation technique to increase the discriminative power of dynamic features with promising results.

Classification of Unlabeled Apps. Table IV lists the different sources of apps for this evaluation. For unlabeled apps with no ground truth available, we also list the detection rates by VirusTotal (including labels for malware and grayware), and the amount of apps that are predicted as malware by

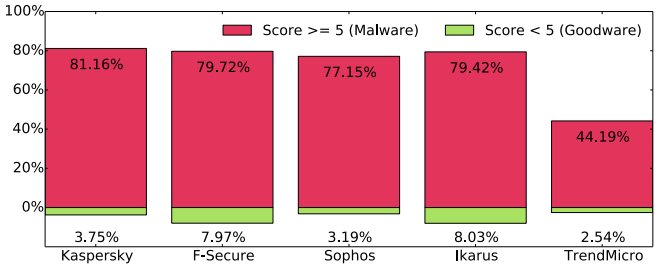


Figure 4: Percentage of apps detected by the top 5 AV scanners as malware or grayware.

each classifier. In order to estimate MARVIN's performance on this data set and as the majority of market apps was crawled before Bouncer was deployed, we investigate the apps from the Google Play Store that MARVIN classified as malware in more detail. For each classifier the majority of high scores are assigned to variants of the *Android.Trojan.IconoSys* family. This assessment is based on certificates reused among apps, SMS-related permissions and sending SMS, and contacting the URLs `blackflyday.com`, `iconosys.com` and `smsreplier.net` during dynamic analysis. Other correct malware detections come from the *Android.Trojan.FakeDoc* and *Plankton* families. MARVIN also flags spyware that requests permissions to send and receive SMS, leaked the IMSI and dynamically registered broadcast receivers for `sms_received` and `new_outgoing_call` events. The majority of benign apps that receive high scores are apps utilizing the aggressive AirPush ad library, that has been banned by Google [61] and that is considered adware by most AV scanners. Also flagged are two mobile AV apps that start a service on startup, load native code, dynamically register broadcast receivers for package installation and uninstallation events, and request permissions to read and send SMS as well as modify the APN settings.

Comparison with AV Results. In order to compare our method to traditional malware detection techniques, we evaluated MARVIN's detection rate on the unlabeled set against the individual AV scanner results provided by VirusTotal. For brevity, we only show the comparison against the 5 best-performing AV scanners in terms of detection rates. Figure 4 illustrates that those scanners detect a maximum of around 80% of apps MARVIN flags as malware (i.e. assigned scores >5). In contrast, the same scanners detect between 2.5% and 8% of apps that receive scores lower than 5 as malware or grayware.

Retraining Strategy. As the malware landscape changes over time, due to new monetization opportunities being exploited, and the never-ending arms race between malware authors and security researchers, MARVIN's classification model will age and will be outdated at some point. Thus, frequent retraining is an absolute necessity. Our retraining intervals are, however, dependent on the availability of ground truth. Apvrille et al. [30] estimated that it takes security researchers up to three months to detect new Android malware in the wild. Our original training data includes malware samples seen until September 2012. Thus, as a worst case scenario we obtained an additional retraining set comprised of 1,134 malicious apps that we randomly selected from apps that were first submitted to VirusTotal in January 2013 and that matched 10 or more AV signatures. With the three months old training data MARVIN is still able to correctly classify 88.54%/91.01% (L_1/L_2-regularized linear classification) of these new apps correctly.

In order to evaluate the retraining effectiveness, we then split the new test data in a training set (70%) and a testing set (30%). When training the classifier on the new training set only, MARVIN achieves a detection rate of 96.47% and 94.71% (L_1 and L_2) on the new test set. However, the detection rate on the original testing data deteriorates to 57.48% and 43.18%. When combining the original training data with the new training set MARVIN's detection rate recovers to over 98%. In order to guarantee the optimal performance of our classifier we thus retrain MARVIN regularly with new apps as soon as new AV labels become available while still keeping part of older training apps in the training set.

With another retraining set comprised of both malware and goodware we further evaluate how malicious and as benign features shift over longer periods of time. We randomly selected 2,874 malicious apps from new submissions to VirusTotal from January 2013 to May 2014. Additionally, we crawled 7,626 of the most popular apps from Google Play in April and May 2014. The proportion of malware to goodware (27% to 73%) in this set reflects the increasing number of malware submissions to ANDRUBIS. On this data set the L_1-regularized classifier correctly classified 71.50% of malware and 70.52% of goodware, while the L_2-regularized classifier proved superior by accurately classifying 77.59% of malware and 95.16% of goodware. Note that in this pathological case no retraining was performed for over one year. Yet, these results indicate that benign features are more stable over time and retraining with additional goodware is necessary less frequently than with malware. When extending the original training data with the apps from this retraining set MARVIN achieves a true positive rate of 96.52%/97.80% (L_1/L_2) and a true negative rate of 99.78%/99.61%, demonstrating the long-term practicality of MARVIN without requiring any adjustments to the feature extraction process.

Table V: Features considered for SVM classification.

Data Set	Static F-Score #	Static F-Score max	Dynamic F-Score #	Dynamic F-Score max	Non-Zero F-Score #	Non-Zero F-Score %
Feature Selection Set	32,022	1.52	26,299	0.27	58,321	11.74%
Labeled Training Set	214,487	1.24	95,239	0.31	309,726	62.33%
All Labeled Samples	279,685	1.24	124,976	0.32	404,661	81.43%

Table VI: Features considered for linear classification.

	Static Weights MW #	Static Weights MW mean	Static Weights GW #	Static Weights GW mean	Dynamic Weights MW #	Dynamic Weights MW mean	Dynamic Weights GW #	Dynamic Weights GW mean	Non-Zero Weight #	Non-Zero Weight %
L_1	783	1.19	394	0.50	272	1.04	120	0.29	1,569	0.32%
L_2	6,558	0.15	208,161	0.01	23,638	0.01	71,624	0.01	309,981	62.38%

C. DISTINGUISHING MALWARE FEATURES

Finally, we examine the number and nature of the most decisive features when distinguishing benign from malicious apps to better understand MARVIN's risk assessment.

Number of Relevant Features. For classification with the SVM the F-scores indicate how discriminative the individual features are. We calculate the F-score for three different subsets of the labeled data set: the subset exclusively used for feature selection of the SVM, the subset used for training the SVM, and the labeled data set as a whole. Additionally, we calculate

the percentage of features out of the whole evaluation feature space that are expressed in a particular subset and judged to be at least marginally relevant by this measure. The results can be found in Table V: On the feature selection set, 11.74% of the feature space is assigned an F-score > 0, on the labeled training set 62.33%, and on all labeled data 81.43%. Static analysis features are assigned the highest ranking scores, with `send_sms`, `receive_sms`, and `read_phone_state` permissions being the top three. Related work by Felt et al. [27] also determined that those three permissions were characteristic of malware. However, more dynamic analysis features than static analysis features were assigned high scores. This is also reflected by the feature subset selected by the feature selection: It predicts the highest accuracy for a set of the 27,808 highest ranked features with a minimum F-score of 0.000109. 18,335 of those features are dynamic, 9,473 are static. We use this subset of features for the evaluation of the SVM detailed in this section.

For the linear classifiers, their model contains the weights assigned to each feature and thus its importance in the classifying process. Weights can either be negative or positive, depending on which class a feature is an indicator of. Features that are assigned a weight of zero have no part in the decision making process. As already mentioned in Section IV-B the weighting strategy differs greatly for L_1- and L_2-regularized linear classification. This fact is illustrated in Table VI: L_1 regularization considers only 0.32% of all features for classification, while L_2 regularization considers 62.38% of features relevant. Nevertheless, both strategies yield comparable results on our test set, although L_2 regularization performs slightly better on the previously unseen Genome Project and noticeably better with outdated training data after several months. Both strategies performing comparably well on the data sets that are very similar to the training data, but L_2 regularization performing better on unrelated test sets suggests that L_1 regularization is underfitting the training data and the resulting model is too simple.

Relevant Feature Categories. In order to determine the categories of features that are the most distinguishing between malware and goodware, we calculate the mean of all F-scores and of all weights for each category of features. One drawback of the F-score is that it only outputs the degree of discrimination a feature provides, but it is no indication in favor of which class. In the case of feature weights, however, we can assume that features with a positive weight are indicative of the positive class, i.e. malware in our case. We can also highlight those features in the analysis report to provide an explanation on why a certain app was classified as malicious by MARVIN.

Among the usual suspects, when it comes to features characteristic of malware, are a number of features that are unique to Android apps, such as required permissions, sending SMS, leaking information, and features related to the dynamic loading of code. The high ranking of SMS related activities and permissions among common malware features is not surprising as sending premium SMS is a popular monetization vector of Android malware [57]. The classification that dynamic code loading is as an indicator for maliciousness is also in line with observations in related work [33,57]. Other characteristics for malware are features extracted from the certificates used to sign the apps. Our results confirm that malware authors often reuse the same certificate across variants, or use public testing and debug certificates as stated in previous work [30]. Other distinguishing features are network-related activities that have already been successfully applied to the classification of x86 malware in the past. Looking at the list of hosts contacted during dynamic analysis, especially the top-level domains `.cn` (China), `.ru` (Russia), `.in` (India), and `.biz` are more frequently contacted by malware, while `.kr` (Korea), `.de` (Germany), `.com`, and `.mobi` domains are more likely to be contacted by goodware. The individual hosts among the highest ranking features include C&C servers associated with Android malware, such as `client.a1b2c3d4e5.in` for *Android.Frogonal* and `depot.bulks.jp` for *Android.Dougalek*.

Applications for Distinguishing Features. The result of MARVIN provides users with the malice score of an app and a detailed report on the app's static and dynamic analysis features. With the knowledge of how individual features rank in the decision making process of a classifier, we are able to highlight high ranking features and thus give users an indication as to why an app received a certain malice score. Another application of the feature ranking produced by MARVIN is the integration of highly ranked features in blacklists: MARVIN can reveal the hosts frequently contacted by malware and thus provide a candidate set for URL blacklists. Similarly, MARVIN can also disclose the certificates that are commonly misused by malware authors to sign their apps and which, therefore, should not be trusted.

VI. LIMITATIONS

Similar to other approaches leveraging dynamic analysis or machine learning to analyze and distinguish malicious from benign applications, MARVIN has some limitations.

One shortcoming of our method to check apps on real devices is the way they are submitted. The mobile front-end accesses the APK archive only after it was successfully installed on the target device. If the user decides to try the downloaded app before submitting it to MARVIN, chances are high that an infection already took place. To overcome this problem, our app would need to intercept downloads for apps from the official market. Although no official API for the Google Play Store is available, the proprietary protocol used for downloading apps has already been successfully reverse engineered [62]. To intercept installs from third-party markets, our app would need an instrumentation for each available market. On the other hand, apps are not directly installed if they are downloaded from an alternate source. Instead, the app is stored as an APK file on the device first. At this point, it can be easily analyzed and rated by MARVIN without the possibility of infecting the target device.

Another limitation of any VM-based dynamic analysis approach is evasion. Possibilities reach from iterating certain device properties to reveal the underlying virtualization technology, to exploiting specific properties of emulated code within virtual machines to detect analysis environments [48,49,63]. However, MARVIN does not rely on features extracted from dynamic analysis alone and static analysis features proved highly effective during our evaluation. Therefore, we argue that this is currently not a problem and that it can be addressed in the future by building a more transparent analysis environment.

When malware uses native code to perform malicious activities, we currently only detect that native code was loaded, through both static code analysis and during dynamic analysis. We plan to also incorporate the behavior of the native code by using system-level events to enrich dynamic features (see Section VII).

Finally, malware authors who are aware of the way MARVIN extracts features might try to subvert the classification by attempting a mimicry attack. To this end they would need to make their malware express as many goodware and as little malware specific features as possible. However, they would then have to keep up with the pace of our retraining and hide all malicious behavior from the dynamic analysis environment to achieve this. Furthermore, dynamic analysis is more resilient to mimicry attacks than static analysis and often able to discover the malicious behavior regardless.

VII. FUTURE WORK

One extension we plan to incorporate is to include system-level events as part of the behavioral aspects (dynamic features) of an app. In our evaluation we show that static analysis features are equally, or even more decisive to create the model for an accurate assessment than dynamic analysis features (but not both). Incorporating system calls into our feature space can improve the behavioral models and, in turn, lead to more accurate results for the combined system. Furthermore, by using system-level events, malware utilizing root exploits can be identified and characterized more precisely. We are also currently exploring how MARVIN could benefit from more intelligent user interactions than the current state-of-the-art user interface stimulation in its underlying dynamic analysis environment.

Furthermore, application markets provide a wealth of information about the offered apps. Therefore, we plan to integrate meta-information from markets, such as the number of downloads, user ratings, other apps by the same author, and the lifetime of an app in the market as additional features.

As another direction for future work we plan to evaluate the practicality of our feature set for other applications such as the detection of malicious repackaged apps. MARVIN could raise alarms for apps that have a large overlap of features but are assigned very different malice scores in our ranking.

VIII. CONCLUSION

In this paper, we presented MARVIN, an effective and efficient analysis tool to assess the maliciousness of previously unknown Android apps. MARVIN utilizes machine learning techniques to classify apps based on a rich and comprehensive feature set extracted from static and dynamic analysis of a set of known malicious and benign apps. Our evaluation showed that MARVIN is capable creating an accurate snapshot of malware behavior that it can leverage to assess the risk associated with apps under investigation accurately and comprehensively. In our large-scale data set it correctly identifies 98.24% of malicious apps with less than 0.04% false positives. Furthermore, we showed that it can be efficiently retrained to maintain its detection accuracy in the long term and to adapt to changes in the malware landscape and analysis evasion techniques.

To provide an appropriate end-user experience helpful for novice and expert users alike, MARVIN accepts submissions and displays analysis results through a web interface and a dedicated mobile app. In addition to the added benefit for ordinary users, MARVIN provides malware analysts with the means to pre-select samples for manual investigation. By setting a report threshold, our system is able to filter malware candidates with high precision and provide detailed information about their static analysis features and dynamic behavior to facilitate further (manual) analysis.

ACKNOWLEDGMENTS

We thank VirusTotal for the service they provided for our evaluation. Furthermore, we thank Kevin Borgolte and Abdelberi Chaabane for their valuable feedback. Finally, we thank Alexej Strelzow for his support in developing the mobile app.

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n. 257007 (SysSec) and from the FFG – Austrian Research Promotion under grant COMET K1. This work also has been carried out within the scope of u'smile, the Josef Ressel Center for User-Friendly Secure Mobile Environments. We gratefully acknowledge funding and support by the Christian Doppler Gesellschaft, A1 Telekom Austria AG, Drei-Banken-EDV GmbH, LG Nexera Business Solutions AG, and NXP Semiconductors Austria GmbH.

REFERENCES

[1] IDC, "Worldwide Smartphone Shipments Edge Past 300 Million Units in the Second Quarter; Android and iOS Devices Account for 96% of the Global Market, According to IDC ," http://www.idc.com/getdoc.jsp?containerId=prUS25037214, 2014.

[2] E. Protalinski, "A first: Hacked sites with android drive-by download malware," http://www.zdnet.com/blog/security/a-first-hacked-sites-with-android-drive-by-download-malware/11810, 2012.

[3] F-Secure, "Threat Report H2 2013," http://www.f-secure.com/static/doc/labs_global/Research/Threat_Report_H2_2013.pdf, 2014.

[4] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis, "AndRadar: fast discovery of android applications in alternative markets," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2014.

[5] I. Asrar, "Android.Dropdialer Identified on Google Play," http://www.symantec.com/connect/blogs/androiddropdialer-identified-google-play, 2012.

[6] F. Chytry, "Google Play: Whats the newest threat on the official Android market?" http://blog.avast.com/2014/03/07/google-play-whats-the-newest-threat-on-the-official-android-market, March 2014.

[7] S. Hirst, "Lookout Discovers SocialPath Malware in Google Play Store," https://vpncreative.net/2015/01/10/lookout-socialpath-malware-google-play, 2015.

[8] H. Lockheimer, "Android and Security," http://googlemobile.blogspot.com/2012/02/android-and-security.html, 2012.

[9] J. Oberheide and C. Miller, "Dissecting the Android Bouncer," in *SummerCon*, 2012.

[10] N. J. Percoco and S. Schulte, "Adventures in Bouncerland," in *Black Hat USA*, 2012.

[11] X. Jiang, "An Evaluation of the Application ("App") Verification Service in Android 4.2," http://www.cs.ncsu.edu/faculty/jiang/appverify, 2012.

[12] P. H. Chia, Y. Yamamoto, and N. Asokan, "Is This App Safe? A Large Scale Study on Application Permissions and Risk Signals," in *International Conference on World Wide Web (WWW)*, 2012.

[13] T. Micro, "The Mobile Cybercriminal Underground Market in China," http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-the-mobile-cybercriminal-underground-market-in-china.pdf, 2014.

[14] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[15] J. Cheng, S. H. Wong, H. Yang, and S. Lu, "SmartSiren: Virus Detection and Alert for Smartphones," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2007.

[16] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian, "Virtualized In-cloud Security Services for Mobile Devices," in *Workshop on Virtualization in Mobile Computing (MobiVirt)*, 2008.

[17] M. Endler, "Does Mobile Antivirus Software Really Protect Smartphones?" http://www.informationweek.com/security/antivirus/does-mobile-antivirus-software-really-pr/240008673, 2012.

[18] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[19] M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario, "Automated Classification and Analysis of Internet Malware," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[20] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in *Annual Network & Distributed System Security Symposium (NDSS)*, 2009.

[21] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and Classification of Malware Behavior," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[22] M. Neugschwandtner, P. Milani Comparetti, G. Jacob, and C. Kruegel, "ForeCast: Skimming off the Malware Cream," in *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[23] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: Versatile Protection for Smartphones," in *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[24] C. Jarabek, D. Barrera, and J. Aycock, "ThinAV: Truly Lightweight Mobile Cloud-based Anti-malware," in *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[25] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using Probabilistic Generative Models For Ranking Risks of Android Apps," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[26] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[27] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A Survey of Mobile Malware in the Wild," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[28] S. Rosen, Z. Qian, and Z. M. Mao, "AppProfiler: A Flexible Method of Exposing Privacy-Related Behavior in Android Applications to End Users," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[29] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot, "Understanding and Improving App Installation Security Mechanisms Through Empirical Analysis of Android," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[30] A. Apvrille and T. Strazzere, "Reducing the Window of Opportunity for Android Malware: Gotta catch 'em all," *Journal in Computer Virology*, vol. 8, no. 1-2, pp. 61–71, 2012.

[31] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "MAST: Triage for Market-scale Mobile Malware Analysis," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2013.

[32] H. Zhu, H. Xiong, Y. Ge, and E. Chen, "Mobile App Recommendations with Security and Privacy Awareness," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2014.

[33] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and Accurate Zero-day Android Malware Detection," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[34] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android," in *International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2013.

[35] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket," in *Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[36] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[37] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android Malware Detection Through Manifest and API Calls Tracing," in *Asia Joint Conference on Information Security*, 2012.

[38] B. Sanz, I. Santos, C. Laorden, X. U.-P. P. Bringas, and G. Alvarez, "PUMA: Permission Usage to detect Malware in Android," in *Advances in Intelligent Systems and Computing (AISC)*, 2012.

[39] J. Sahs and L. Khan, "A Machine Learning Approach to Android Malware Detection," in *European Intelligence and Security Informatics Conference (EISIC)*, 2012.

[40] A. Shabtai, U. Kananov, Y. Elovici, C. Glezer, and Y. Weiss, ""Andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, pp. 161–190, 1 2012.

[41] B. Amos, H. A. Turner, and J. White, "Applying Machine Learning Classifiers to Dynamic Android Malware Detection at Scale," in *International Conference on Wireless Communications and Mobile Computing (IWCMC)*, 2013.

[42] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-Based Malware Detection System for Android," in *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[43] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Annual Network & Distributed System Security Symposium (NDSS)*, 2012.

[44] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying Android malware using dynamically obtained features," *Journal of Computer Virology and Hacking Techniques*, 2014.

[45] W.-C. Wu and S.-H. Hung, "DroidDolphin: A Dynamic Android Malware Detection Framework Using Big Data and Machine Learning," in *Conference on Research in Adaptive and Convergent Systems (RACS)*, 2014.

[46] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[47] A. Aprville and A. Albertini, "Hide Android Applications in Images," Black Hat Europe, 2014.

[48] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware," in *European Workshop on System Security (EuroSec)*, 2014.

[49] T. Vidas and N. Christin, "Evading Android Runtime Analysis via Sandbox Detection," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[50] Google, "Android Developers: Signing Your Applications," http://developer.android.com/tools/publishing/app-signing.html.

[51] N. Šrndić and P. Laskov, "Practical Evasion of a Learning-Based Classifier: A Case Study," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[52] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, "From Throw-away Traffic to Bots: Detecting the Rise of DGA-based Malware," in *USENIX Security Symposium*, 2012.

[53] A. Y. Ng, "Feature selection, L1 vs. L2 regularization, and rotational invariance," in *International Conference on Machine Learning (ICML)*, 2004.

[54] Y. W. Chen and C. J. Lin, "Combining SVMs with Various Feature Selection Strategies," in *Feature Extraction, Foundations and Applications*. Springer, 2006.

[55] "VirusTotal," http://www.virustotal.com.

[56] F-Secure, "Mobile Threat Report Q2 2012," http://www.f-secure.com/weblog/archives/MobileThreatReport_Q2_2012.pdf, 2012.

[57] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[58] "Contagio," http://contagiominidump.blogspot.com.

[59] H. T. T. Truong, E. Lagerspetz, P. Nurmi, A. J. Oliner, S. Tarkoma, N. Asokan, and S. Bhattacharya, "The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators," in *International Conference on World Wide Web (WWW)*, 2014.

[60] "AppBrain Stats," http://www.appbrain.com/stats/number-of-android-apps, (Retrieved: April 27, 2015).

[61] L. Spradlin, "Google Updates Play Store Developer Policy, Puts The Smack Down On Intrusive Advertising," http://www.androidpolice.com/2012/07/31/google-updates-play-store-developer-policy-puts-the-smack-down-on-intrusive-advertising-say-goodbye-to-airpush-and-its-cohorts/, 2012.

[62] T. Strazzere, "Downloading market applications without the vending app," http://www.strazzere.com/blog/2009/09/downloading-market-applications-without-the-vending-app, 2009.

[63] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware," in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008.