Android Malware Detection Based on System Calls

Marko Dimjašević, Simone Atzeni,
Ivo Ugrina, Zvonimir Rakamarić

UUCS-15-003

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

May 15, 2015

Abstract

With Android being the most widespread mobile platform, protecting it against malicious applications is essential. Android users typically install applications from large remote repositories, which provides ample opportunities for malicious newcomers. In this paper, we propose a simple, and yet highly effective technique for detecting malicious Android applications on a repository level. Our technique performs automatic classification based on tracking system calls while applications are executed in a sandbox environment. We implemented the technique in a tool called MALINE, and performed extensive empirical evaluation on a suite of around 12,000 applications. The evaluation yields an overall detection accuracy of 93% with a 5% benign application classification error, while results are improved to a 96% detection accuracy with up-sampling. This indicates that our technique is viable to be used in practice. Finally, we show that even simplistic feature choices are highly effective, suggesting that more heavyweight approaches should be thoroughly (re)evaluated.

Android Malware Detection Based on System Calls

Marko Dimjašević, Simone Atzeni, Zvonimir Rakamarić
University of Utah, USA
{marko,simone,zvonimir}@cs.utah.edu

Ivo Ugrina
University of Zagreb, Croatia
ivo@iugrina.com

***Abstract*—With Android being the most widespread mobile platform, protecting it against malicious applications is essential. Android users typically install applications from large remote repositories, which provides ample opportunities for malicious newcomers. In this paper, we propose a simple, and yet highly effective technique for detecting malicious Android applications on a repository level. Our technique performs automatic classification based on tracking system calls while applications are executed in a sandbox environment. We implemented the technique in a tool called MALINE, and performed extensive empirical evaluation on a suite of around 12,000 applications. The evaluation yields an overall detection accuracy of 93% with a 5% benign application classification error, while results are improved to a 96% detection accuracy with up-sampling. This indicates that our technique is viable to be used in practice. Finally, we show that even simplistic feature choices are highly effective, suggesting that more heavyweight approaches should be thoroughly (re)evaluated.**

I. Introduction

The global market for mobile devices has exploded in the past several years, and according to some estimates the number of smartphone users alone reached 1.7 billion worldwide in 2014. Android is the most popular mobile platform, holding nearly 85% of the global smartphone market share [1]. One of the main advantages of mobile devices such as smartphones is that they allow for numerous customizations and extensions through downloading and installing applications from public application markets. The largest of such markets (e.g., Google Play, Apple App Store) have more than one million applications available for download each, and there are more than 100 billion mobile device applications installed worldwide.

This clearly provides a fertile environment for malicious activities, including development and distribution of malware. A recent study [2] estimates that the total amount of malware across all mobile platforms grew exponentially at the rate of 600% between 03/2012 and 03/2013. Around 92% of malware applications found in this study target Android. In a related study [3], similar statistics are reported — the number of malicious applications in the Google Play store grew around 400% from 2011 to 2013, while at the same time the number of malicious applications removed annually by Google has dropped from 60% in 2011 to 23% in 2013. Due to the sharp increase in the total amount of malware, the percentage of removed malware dropped significantly despite the fact that the absolute number actually increased from roughly 7,000 in 2011 to nearly 10,000 in 2013. While companies such as Google regularly scan their official application market using advanced, proprietary tools, this process is still often ineffective as the above numbers illustrate. There are also unofficial, open markets where often no scanning is being performed, partially because there is a lack of solid freely available solutions and tools. As a consequence, Android malware detection has been an active area of research in the past several years, both in industry and academia. Currently, published approaches can be broadly categorized into manual expert-based approaches, and automatic static- or dynamic-analysis-based techniques.

Expert-based approaches detect malware by relying on manually specified malware features, such as requested permissions [4] or application signatures [5], [6]. This requires significant manual effort by an expert user, is often easy to circumvent by malware writers, and targets existing, specific types of malware, thereby not providing protection from evolving malicious applications.

Static-analysis-based techniques typically search for similarities to known malware. This often works well in practice since new malware is typically just a variation of the existing ones. Several such techniques look for code variations [7], [8], which becomes ineffective when faced with advanced code obfuscation techniques. Hence, researchers have been exploring more high-level properties of code that can be extracted statically, such as call graphs [9]. Unfortunately, even those approaches can be evaded by leveraging well-known drawbacks of static analysis. For example, generated call graphs are typically over-approximations, and hence can be obfuscated by adding many dummy, unreachable function calls. In addition, native code is hard to analyze statically, and hence malicious behavior can be hidden there.

Dynamic analysis techniques typically run applications in a sandbox environment or on real devices in order to extract information about the application behavior. The extracted behavior information is then automatically analyzed for malicious behavior using various techniques, such as machine learning. Recent techniques is this category often observe application behavior by tracing system calls in a virtualized environment [10]–[12]. However, both static analysis and dynamic analysis proponents made various claims, often contradicting ones — including claims that are based on questionably designed experiments — on effectiveness of malware detection based on system calls.

In this paper, we propose a dynamic Android malware detection approach based on tracking system calls, and we implement it as a free and open-source tool called MALINE. Our work was initially inspired by a similar approach proposed for desktop malware detection [13], albeit we provide simpler feature encodings, an Android-specific tool flow, and extensive empirical evaluation. We provide several encodings of behavior fingerprints of applications into features for subsequent classification. We performed an extensive empirical evaluation on a set of more than 12,000 Android applications. We analyze how the quality of malware classifiers is affected across several dimensions, including the choice of an encoding of system calls into features, the relative sizes of benign and malicious data sets used in experiments, the choice of a classification algorithm, and the size and type of inputs that drive a dynamic analysis. Furthermore, we show that the structure of system call sequences observed during application executions conveys in itself a lot of information about application behaviors. Our evaluation sheds light on several such aspects, and shows that the proposed combinations can be effective: our approach yields overall detection accuracy of 93% with a 5% benign application classification error. Finally, we provide guidelines for domain experts when making choices on malware detection tools for Android, such as MALINE.

Our approach provides several key benefits. By guarding the users at the repository level, a malicious application is detected early and before it is made publicly available for installation. This saves scarce energy resources on the devices by delegating the detection task to a trusted remote party, while at the same time protecting users' data, privacy, and payment accounts. System call monitoring is out of reach of malicious applications, i.e., they cannot affect the monitoring process. Hence, our analysis that relies on monitoring system calls happens with higher privileges than those of malicious applications. In addition, tracking system calls entering the kernel (and not calls at the Java library level) enables us to capture malicious behavior potentially hidden in native code. Since our approach is based on coupling of a dynamic analysis with classification based on machine learning, it is completely automatic. We require no source code, and we capture dynamic behavior of applications as opposed to their code properties such as call graphs; hence, our approach is mostly immune to common, simple obfuscation techniques. The advantages of our approach make it complementary to many existing approaches, such as the ones based on static analysis.

Our contributions are summarized as follows:

- We propose a completely automatic approach to Android malware detection on the application repository level using system calls tracking and classification based on machine learning, including a novel heuristics-based encoding of system calls into features.
- We implement the approach in a tool called MALINE, and perform extensive empirical evaluation on more than 12,000 applications. We show that MALINE effectively discovers malware with a very low rate of false positives.
- We compare several feature extraction strategies and classifiers. In particular, we show that the effectiveness of even very simplistic feature choices (e.g., frequency of system calls) is comparable to much more heavyweight approaches. Hence, our results provide a solid baseline and guidance for future research in this area.

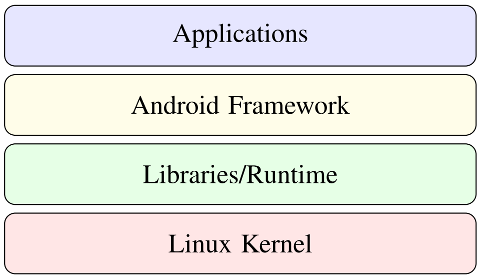

Fig. 1: Abstraction layers of the Android architecture.

II. Preliminaries

A. System Calls

A system call is a mechanism for a program to request a service from the underlying operating system's kernel. In Android, system calls are created by information flowing through a multi-layered architecture depicted in Figure 1. For example, an Android text messaging application, located at the highest level of the architecture, might receive a user request to send an SMS message. The request is transformed into a request to the Telephony Manager service in the Application Framework. Next, the Android runtime receives the request from the service, and it executes it in the Dalvik Virtual Machine.[1] The execution transforms it into a collection of library calls, which eventually result in multiple system calls being made to the Linux kernel. One of the system calls will be `sendmsg`:

```
sendmsg(int sockfd, const struct msghdr* msg,
        unsigned int flags)
```

It is a system call used to send a message on a socket. The generated sequence of system calls represents a low-level equivalent of the SMS message being sent in the application at the highest level of abstraction. Information flows in the opposite direction in a similar fashion.

B. Machine Learning

Our malware detection problem is an instance of a *classification problem* in machine learning, and is solved using a *classifier* algorithm. More specifically, it is an example of a *binary classification problem* since it explores connections between the behavior of an application and its goodware/malware (only two choices) label. The two groups are commonly called a *positive* and a *negative* group. If a positive example (e.g., an application in our case) is classified into the positive (resp., negative) group, we obtained a *true positive/TP* (resp., *false negative/FN*). Analogously, we define *true negative/TN* and *false positive/FP*. Table I gives standard measures of the quality of classification prediction used in machine learning based on these terms.

[1]As of Android version 5.0, the Dalvik Virtual Machine is replaced with an application runtime environment called ART.

measure	formula
accuracy, recognition rate	$\frac{TP+TN}{P+N}$
errorrate, misclassification rate	$\frac{FP+FN}{P+N}$
sensitivity, true positive rate, recall	$\frac{TP}{P}$
specificity, true negative rate	$\frac{TN}{N}$
precision	$\frac{TP}{TP+FP}$

TABLE I: Standard measures of the quality of classifiers. P (resp, N) is the number of positive (resp., negative) examples.

Classification is usually conducted through individual measurable heuristic properties of a phenomenon being investigated (e.g., height of people, their weight, a number of system calls in one run of an Android application). Such properties are called *features*, and a set of features of a given object is often represented as a feature vector. Feature vectors are stored in a feature matrix, where every row represents one feature vector.

More about machine and statistical learning can be found in related literature [14], [15].

III. Our Approach

Our approach is a three-phase analysis, as illustrated in Figure 2. The first phase is a dynamic analysis where we track system calls during execution of an application in a sandbox environment and record them into a log file. In the second phase, we encode the generated log files into feature vectors according to several representations we define. The last phase takes all such feature vectors and applies machine learning in order to learn to discriminate benign from malicious applications.

A. Dynamic Analysis Phase

As our approach is based on concrete executions of applications, the first phase tracks and logs events at the operating system level that an application causes while being executed in a sandbox environment. The generated event logs serve as a basis for the subsequent phases of our analysis. Unlike numerous static analysis techniques, this approach reasons only about events pertaining to the application that are actually observed in the operating system.

A user's interaction with Android through an application results in events being generated at the operating system level, which are rendered as system calls. In our work, we automatically emulate this interaction as explained in detail in Section IV. For that reason, we execute every application in a sandbox environment and observe resulting system calls in a chronological order, from the very beginning until the end of its usage. The output of this phase is a log file containing chronologically ordered system calls: every line consists of a time stamp, the name of the system call, its input values, and the return value, if any. Having the system calls recorded chronologically enables us to construct various feature vectors that characterize the application's behavior with different levels of precision, as explained in the next section.

More formally, let $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ be a set of system call names containing all the system calls available in the Android operating system for a given processor architecture. Then a system call sequence σ of length m, representing the chronological sequence of recorded system calls in a log file, is a sequence of instances of system calls $\sigma = (q_1, q_2, \ldots, q_m)$, where $q_i \in \mathcal{S}$ is the ith observed system call in the log file. Such call sequences are passed to the feature extraction phase.

B. Feature Extraction Phase

As explained earlier, how features are picked for the feature vector is important for the machine learning classification task. Therefore, we consider two representations for generating a feature vector from a system call sequence σ. Our simpler representation is concerned with how often a system call happens, while our richer representation encodes information about dependencies between system calls. Both representations ignore system call information other than their names and sequence numbers (e.g., invocation time, input and output values), as it can be seen from the definition of σ. Once we compute a feature vector $\mathbf{x}$ for every application under analysis according to a chosen representation, we form a feature matrix by joining the feature vectors such that every row of the matrix corresponds to one feature vector.

1) System Call Frequency Representation: How often a system call occurs during an execution of an application carries information about its behavior [16]. A class of applications might be using a particular system call more frequently than another class. For example, some applications might be making considerably more I/O operation system calls than known goodware, indicating that the increased use of I/O system calls might be a sign of malicious behavior. Our simple system call frequency representation tries to capture such features.

In this representation, every feature in a feature vector represents the number of occurrences of a system call during an execution of an application. Given a sequence σ, we define a feature vector $\mathbf{x} = [x_1 x_2 \ldots x_{|\mathcal{S}|}]$, where x_i is equal to the frequency (i.e., the number of occurrences) of system call s_i in σ. In experiments in Section V, we use the system call frequency representation as a baseline comparison against the richer representation described next.

2) System Call Dependency Representation: Our system call dependency representation was inspired by previous work that has shown that a program's behavior can be characterized by dependencies formed through information flow between system calls [17]. However, we have not been able to find a tool for Android that would provide us with this information and also scale to analyzing thousands of applications. Hence, we propose a novel scalable representation that attempts to capture such dependencies by employing heuristics. As we show in Section V, even though our representation is simpler than the one based on graph mining and concept analysis from the original work [17], it still produces feature vectors that result in highly accurate malware detection classifiers.

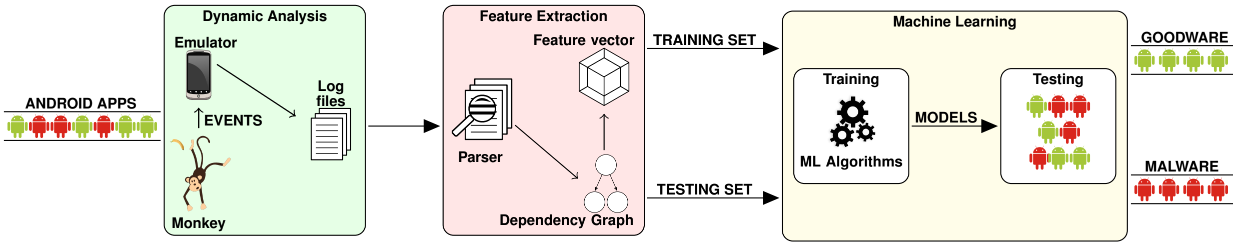

Fig. 2: Maline tool flow divided into three phases.

For a pair of system calls q_i and q_j in a sequence σ, where $i < j$, we define the distance between the calls as $d(q_i, q_j) = j - i$. We then approximate a potential data flow relationship between a pair of system calls using the distance between the calls in a sequence (i.e., log file). For example, if two system calls are adjacent in σ, their distance will be 1. Furthermore, let $w_{g,h}$ denote the weight of a directed edge (s_g, s_h) in a *system call dependency graph* we generate. The system call dependency graph is a complete digraph with the set of vertices being the set of all the system call names $\mathcal{S}$, and hence having $|\mathcal{S}|^2$ edges. Then, $w_{g,h}$ for a sequence σ is computed as:

$$w_{g,h} = \begin{cases} 0, & \text{if } g = h \\ \sum\limits_{\substack{i<j<k, \\ q_i = s_g, q_j = s_h}} \frac{1}{d(q_i, q_j)}, & \text{otherwise} \end{cases}$$

where k is the minimal index such that $q_i = q_k$ and $i < k \leq |\sigma|$. Informally, the closer the pair is in a sequence, the more it contributes to its edge weight in the graph. Hence, instead of explicitly observing a data flow between system calls, our representation captures it implicitly: it is based on a simple observation that the closer a pair of system calls is in a sequence, the more likely it is that there is a data flow between the pair.

From a sequence σ, we compute weights $w_{g,h}$ for every system call pair $(s_g, s_h) \in \mathcal{S}^2$. For g and h such that $w_{g,h} = 0$, we still consider edge (s_g, s_h) to exist, but with the weight of 0. Since each application is executed only once during our dynamic analysis phase, we generate one system call dependency graph per application.

We generate a feature vector $\mathbf{x}$ of an application by taking edge weights from its system call dependency graph. For every directed edge (s_g, s_h) there is a corresponding feature in $\mathbf{x}$, and hence the dimensionality of $\mathbf{x}$ is $|\mathcal{S}|^2$. Given a sequence σ, we define a feature vector $\mathbf{x} = [x_1 x_2 \ldots x_{|\mathcal{S}|^2}]$, where x_i is equal to $w_{g,h}$ such that $i = (g-1) \cdot |\mathcal{S}| + h$.

C. Machine Learning Phase

We use the generated feature vectors for our applications (i.e., feature matrices) together with provided malware/goodware labels to build classifiers. The choice of a classification algorithm for a given problem is not an easy one. Hence, we experimented with several of the most popular and effective ones: support vector machines (SVMs) [18], random forest (RF) [19], LASSO [20], and ridge regularization [21]. Some classification models are tuned using various parameters (e.g., linear kernel in SVMs depends on the regularization parameter c), and hence the quality of classifiers heavily depends on the chosen values for these parameters.

When a probabilistic classifier is used, a threshold that appropriately tunes the trade-off between sensitivity and specificity can be studied using *receiver operating characteristic* (ROC) curves [22]. A ROC curve is created by evaluating the class probabilities for the model across a continuum of thresholds. For each candidate threshold, the resulting sensitivity and specificity are plotted. Generating ROC curves is especially valuable to the users of malware detectors such as ours, since they can use them to fine-tune sensitivity vs. specificity depending on the intended usage. Hence, we generate ROC curves for the most interesting classifiers.

We have chosen to use around 33% samples as malware and the rest as goodware. Although this approach does not generate a perfectly balanced design, (the same number of malware and goodware samples), it tries to represent the goodware population as best as possible while still keeping the high percentage of malware samples and computational costs at a practical level. In addition, we explored what can be achieved by balancing the design through resampling strategies of *up-sampling* (or over-sampling) the minority class and *down-sampling* (or under-sampling) the majority class [23] implemented through bootstrapping.

IV. Implementation

We implemented our approach in a tool called MALINE, and Figure 2 shows its tool flow. The implementation comes as a free software reproducible research environment in order to foster Android security analyses and further research in this area.[2] MALINE heavily utilizes our own build of the Android Software Development Kit (SDK). The SDK includes the Android Emulator, which runs a virtual machine (VM) with the Android operating system. Every application MALINE analyzes is installed, executed, and monitored in the VM. The tool primarily resides on the host machine and relies on the Android Debug Bridge (*adb*) to communicate with the VM.

A. Host and Emulator

MALINE consists of a number of smaller components. We implemented multiple interfaces on the host side, ranging from starting and monitoring an experiment with multiple emulator

[2]The MALINE tool is available at https://github.com/soarlab/maline.

instances running in parallel to machine-learning differences between applications based on the processed data obtained from emulator instances. It is the host side that coordinates and controls all such activities. For example, it creates and starts a pristine installation of Android in an emulator instance, waits for Android to boot, then installs an application in it, starts the application, and waits for it to finish so it can analyze system calls the application has made during its execution.

We use the emulator, which is built on top of QEMU [24], in the dynamic analysis phase of our approach (see Figure 2). For every application we create a pristine sandboxed environment since the emulator enables us to easily create a clean installation of Android. It is important that each application is executed in a clean and controlled environment to make sure nothing is left behind from previous executions and to be able to monitor the execution. Hence, every application's execution is completely independent of executions of all the other analyzed applications.

1) Custom Build of Android SDK: In our implementation, we used the Android 4.4.3 *KitKat* release, which utilizes Android API version 19. However, we have our own build of the Android system implemented on top of the official source code repositories. The main reason for the custom build is to prevent bugs we found in the Android SDK throughout multiple releases. Our build is a compilation of parts from different versions such that the functionality we needed from the SDK works as expected.

Our build also features a modification to the Monkey tool (we describe the tool later) to have better control over experiments. The default Monkey version injects an event into a system queue and moves onto the next event right away, without waiting for the queued event to be executed. However, to make Android more responsive, its developers decided to drop events from the queue when under heavy load. In our experiments, this would mean that events that Monkey injects might be discarded, thereby compromising the dynamic analysis of an application under test. To make sure the Android system does not drop events, we have slightly modified Monkey so that it waits for each event to be executed before proceeding to the next event.

B. Automatic Execution of Applications

In order to scale to thousands of applications, our dynamic analysis phase implements an automatic application execution process. The process starts with making a clean copy of our default virtual machine. The copy contains only what is installed by default in a fresh installation of the Android operating system from the Android Open Source Project. Once the installation boots, we use *adb* to send an application from the host machine to the VM for installation. Next, we start the application and immediately begin tracing system calls related to the operating system process of the application with the *strace* tool. The system calls are recorded into a log file.

We simulate a user interaction with an Android device by injecting both *internal* and *external* events into the emulator. Internal events are sent to the application itself, such as screen clicks, touches, and gestures. We use the Monkey tool [25] as our internal event generator (see Figure 2). It sends a parameterized number of the events to the application, with a 100 ms pause period between consecutive events if applicable[3]. Unlike internal events, which are delivered to the application, external events are delivered to the emulator and include events that come from interacting with an external environment. In our experiments, for external events we focus on generating text messages and location updates only since those are sometimes related to malicious behaviors.

Even though a system calls log file, forming a chronological sequence of low-level operations, contains rich information (e.g., time stamp, input and output values), we preserve only system call names and their order. We stop an application execution when all internal events generated by Monkey are delivered and executed. Finally, the log file is pulled from the VM to the host machine for parsing.

In the next step we apply a feature vector representation, either the system call frequency representation or the system call dependency representation as explained in Section III. The output is a textual feature vector file per log file, i.e. per application, listing all the features. Finally, we combine all the feature vectors into a single matrix where each matrix row corresponds to one feature vector, i.e. one application.

C. Classification

Using the feature matrix generated from logs and previously obtained labels denoting malware/goodware for applications, we proceed with performing a classification. We experimented with several classification algorithms: random forest, SVMs, LASSO, and ridge regression. An implementation of SVMs is based on libSVM [26], while all the other algorithms are implemented in R [27] using the language's libraries [28]. The scripts are heavily parallelized and adjusted to be run on clusters or supercomputers. For example, running a random forest model on a feature matrix from a system call dependency graph sample takes at least 32 GB of RAM in one instance of 5-fold cross-validation.

V. Evaluation

We evaluated MALINE by utilizing a set of 32-core machines with 128 GB of RAM running Ubuntu 12.04. The machines are part of the Emulab testing infrastructure [29]. We wrote scripts to automatize and parallelize all of our experiments, without which our extensive experimental evaluation would not be possible. In our experiments, we use only the x86 Android emulator; the resulting x86 system call set $\mathcal{S}$ has 360 system calls.

A. Input Data Set

In order to evaluate MALINE, we obtained applications from Google Play as goodware and the Drebin dataset [30] as malware. Before we could start using the collected applications

[3]The pause between two consecutive events may not be applicable for actions that are time-dependent, such as screen tapping.

in MALINE, we needed to perform a filtering step. First, we removed applications that we failed to consistently install in the Android emulator. For example, even though every Android application is supposed to be self-contained, some applications had dependencies that were not installed at the time; we do not include such applications in our final data set. Second, we removed all applications that we could not consistently start or that would crash immediately. For example, unlike typical Android applications, application widgets are miniature application views that do not have an Android Activity, and hence they cannot be started from a launch menu.

Applications in the Drebin dataset were collected between August 2010 and October 2012, and filtered by their collectors to contain only malicious applications. The malicious applications come from more than 20 malware families, and are classified based on how an application is installed and activated, or based on its malicious payloads [31]. The aim of our work is not to explore specifics of the families; many other researchers have done that. Therefore, in our experiments, we make no distinction between malicious applications coming from different families. The Drebin dataset contains 5560 malware applications; after filtering, our malicious data set contains 4289 of those applications.

We obtained the benign application data set in February 2014 by utilizing a crawler tool. The tool searched Google Play for free-of-charge applications in all usage categories (e.g., communication, education, music and audio, business), and randomly collected applications with at least 50,000 downloads. To get a good representation of the Google Play applications while keeping the ratio of malware/goodware at an acceptable level for future classification (see Section III-C), we have decided to download roughly three times more goodware applications than the number of the obtained malware applications. Hence, we stopped our crawler at 12789 collected Google Play applications; after filtering, our benign data set contains 8371 of those applications. Note that we make a reasonable assumption that all applications with more than 50,000 downloads are benign; the extent to which the assumption is reasonable has a direct impact on classification results presented in this section. The list of all applications in our input set is published in the MALINE repository.

B. Configurations

We explore effects of several parameters in our experiments. The first parameter is the number of events we inject with Monkey into the emulator during an application execution. The number of events is directly related to the length of the execution. We insert 1, 500, 1000, 2000, and 5000 events. It takes 229 seconds on average (with a standard deviation of 106 seconds) for an application execution with 500 events and 823 ($\pm$816) seconds with 5000 events.[4] That includes the time needed to make a copy of a clean virtual machine, boot it, install the application, run it, and download log files from the virtual machine to the host machine.

The second parameter is a flag indicating if a benign background activity should be present while executing the applications in the emulator. The activity comprises of inserting SMS text messages and location updates into the emulator. We experiment with the activity only in the 500-Monkey-event experiments, while for all the other experiments we include no background activity.

It is important to ensure that consistent sequences of events are generated across executions of all applications. As Monkey generates pseudo-random events, we use the same pseudo-random seed value in all experiments.

C. Experimental Results

The total number of system calls an application makes during its execution directly impacts its feature vector, and potentially the amount of information it carries. Hence, we identified the number of injected events, which directly influences the number of system calls made, as an important metric to track. The number of system calls observed per application in the dynamic analysis phase of an experiment varies greatly. For example, in an experiment with 500 Monkey events it ranges from 0 (for applications that failed to install and are filtered out) to over a million. Most of the applications in this experiment had less than 100,000 system calls in total.

1) Feature Matrices: After the dynamic analysis and feature extraction phases (see Section III) on our filtered input set, MALINE generated 12 different feature matrices. The matrices are based on varying experiment configurations including: 5 event counts (1, 500, 1000, 2000, 5000), 2 system call representations (frequency- and dependency-graph-based), and the inclusion of an optional benign activity (SMS messages and location updates) for experiments with 500 events. We refer to these matrices with X_{rep}^{size}, where $rep \in \{freq, graph\}$ is the used representation of system calls and $size$ is the number of generated events. In addition, we denote an experiment with the benign background activity using $*$.

Obtained feature matrices generated according to the system call dependency representation exhibited high sparsity. This is not surprising since the number of possible system call pairs is 129600. Hence, all columns without a nonzero element were removed from our matrices. Table II gives the dimensions of the obtained matrices and their level of sparsity.

Both the frequency and dependency feature vector representations resulted in different nonzero elements in the feature matrices. However, those differences could have only a small or no impact on the quality of classification, i.e., it might be enough only to observe if something happened encoded as zero/one values. Therefore, we have created additional feature matrices by replacing all nonzero elements with ones hoping to catch the effect of feature matrix structure on the classification.

2) Cross-validated Comparison of Classifiers: Reduced feature matrices (just feature matrices from now on) and goodware/malware labels are input to the classification algorithms we used: support vector machines (SVMs), random forest

[4]The standard deviations are relatively large compared to the averages because some applications crash in the middle of their execution. We take recorded system call traces up to that point as their final execution traces.

	full matrix	reduced matrix	
Type	non-zero (%)	columns	non-zero (%)
X^{1}_{freq}	12.48	118	38.09
$X^{500}_{freq}*$	17.30	137	45.48
X^{500}_{freq}	17.27	138	45.07
X^{1000}_{freq}	17.65	136	46.72
X^{2000}_{freq}	17.93	138	46.79
X^{5000}_{freq}	18.15	136	48.04
X^{1}_{graph}	1.49	11112	17.42
$X^{500}_{graph}*$	3.01	15101	25.83
X^{500}_{graph}	2.99	15170	25.61
X^{1000}_{graph}	3.12	15137	26.79
X^{2000}_{graph}	3.22	15299	27.34
X^{5000}_{graph}	3.29	15262	27.97

TABLE II: Comparison of the number of nonzero elements in the reduced (zero-columns removed) and full feature matrices.

(RF), LASSO, and ridge regression. To avoid possible over-fitting, we employed double 5-fold cross-validation on the set of applications to tune parameters and test models. To enable comparison between different classifiers for different feature matrices, the same folds were used in the model building among different classification models. Prior to building the final model on the whole training set, all classifiers were first tuned by appropriate model selection techniques to derive the best parameters. The SVMs algorithm in particular required an expensive tuning phase: for each dataset we had to run 5-fold cross-validation to find the best C and γ parameters. Hence, we had to run the training and testing phases with different values of C (ranging from 2^{-5} to 2^{15}) and γ (ranging from 2^{-15} to 2^{3}) for the 5 different splits of training and testing set. In the end, the best kernel to use with SVMs was the Radial Basis Function (RBF) kernel.

The built classifiers were then validated on the appropriate test sets. Figure 3 shows measures of the quality of prediction (i.e., accuracy, sensitivity, specificity, and precision; see Table I) averaged between cross-validation folds for different classifiers. For exact values that are graphically presented in Figure 3 consult Table III in the Appendix. The threshold for probabilistic classifiers was set at the usual level of 0.5. Since changes to this threshold can have an effect on the sensitivity and the specificity of classifiers, a usual representation of the effect of these changes is given by ROC curves (see Figure 4 for an example). In the paper we give ROC curves only for the random forest models (as the best classifiers judging from the cross-validated comparison) with the largest number of events (5000).

As it can be seen from Figure 3, 1-event quality measures are consistently the worst in each category, often with a large margin. This indicates the importance of leveraging during classification the information gathered while driving an application using random events. Moreover, the random forest algorithm consistently outperforms all other algorithms across the four quality measures. In the case where feature matrices have weights instead of zeros and ones, it shows only small variations across all the input parameters, i.e., the number of events inserted by Monkey, whether there was any benign background activity, and the chosen feature vector representation. Other classification algorithms perform better on the dependency than on the frequency representation. Of the other algorithms, the SVMs algorithm is most affected by the presence of the background activity, giving worse sensitivity with the presence, but on the other hand giving better specificity.

When the weights in the feature matrices are replaced with zeros and ones, thereby focusing on the structure of the features and not their values (see Section V-C1), all the algorithms consistently perform better on the dependency than on the frequency feature vector representation. However, a comparison within an algorithm based on the weights or zeros and ones in the feature matrices is not straightforward. Random forest clearly performs worse when zeros and ones are used in the feature matrices. LASSO and ridge typically perform better in all the quality measures apart from sensitivity for the zeros and ones compared to the weights. We have not performed the same comparison for 1-event experiments but for random forest due to significantly higher resource demands of the algorithms; we plan to have that data ready as well for the final version of the paper.

If a domain expert in Android malware detection is considering to apply MALINE in practice, there are several practical lessons to be learned from Figure 3. The expert can choose to use only the random forest algorithm as it consistently provides the best outcomes across all the quality measures. To reduce the time needed to dynamically analyze an application, it suffices to provide 500 Monkey events as an application execution driver. Furthermore, the presence of the benign background activity does not make much of a difference. On the other hand, to provide few execution-driving events to an application does not suffice. Finally, if the time needed to learn a classifier is crucial and the expert is willing to sacrifice sensitivity, the expert can choose the frequency feature vector representation since it yields almost as good results as the dependency one, but with far smaller feature vectors, which implies a much smaller demand on computational resources.

Figure 4 shows that there is not much variability between 5 different folds from the cross-validation of the best-performing algorithm, namely random forest. This indicates a high stability of the random forest model on the input data set regardless of the choice of training and test sets. It is up to the domain expert to make the trade-off choice in tuning a classifier towards either high sensitivity or specificity. The choice is directly related to the cost of having false positives, the benefits of having more true positives, etc. For example, the domain expert may choose the dependency graph feature vector representation and fix the desired specificity level to 95%; from the right-hand side ROC curve in Figure 4 it follows that the sensitivity level would be around 93%. Figure 6 in Appendix A gives two more ROC curves for a random forest classifier with up-sampling, where the variability is even smaller and the overall performance of the classifier is better than when no up-sampling is performed.

3) Exploring the Effect of Matrix Sparsity: Sparsity of feature matrices can sometimes lead to overfitting. Although we significantly reduce the sparsity with the removal of columns with all zeros, this just removes non-informative features and sparsity is still relatively high (25% for graph representations). To be sure that the effect seen in the cross-validation comparison is real, we performed additional exploration by adopting the idea of permutation tests [32].

Due to prohibitively high computational costs, only one classification model was used to explore the effect of sparsity. We chose the random forest classifier, since it gave the best results on the cross-validation comparison, and the 5000-event matrices. Prior to building a classifier, we permute application labels. On permuted labels the same procedure (5-fold cross-validation) as in Section V-C2 was used, thus obtaining quality of prediction on the permuted sample. This procedure is repeated 1000 times. Average accuracies of the obtained classifiers were compared to the accuracy of the RF model from Section V-C2 and they were all significantly lower — the best is at 83% for the system call dependency representation. Although 1000 simulations is not much in permutation models, it still reduces the probability of accidentally obtaining high quality results just because of sparsity.

4) Exploring the Effect of Unbalanced Design: Since the number of malware applications in our input set is half the number of goodware, we have an *unbalanced design*. In order to explore if we could get better results using balanced designs (the same number of malware and goodware), we employed down-sampling and up-sampling through bootstrapping. We used only the random forest classifier on different feature matrices to keep computational costs feasible.

Up- and down-sampling exhibited the same effect on the quality of prediction for all feature matrices: increasing sensitivity at the cost of decreasing specificity. This does not come as a surprise since we have equated the number of malware and goodware applications, thereby giving larger weights to malware applications in the model build. However, the overall accuracy for models with down-sampling was lower than for the unbalanced model, while for models with up-sampling it was higher (up to 96.5% accuracy with a 98% sensitivity and 95% specificity). To explore the stability of results under down- and up-sampling, these methods were repeated 10 times; the standard deviation of accuracies between repeats (on percentage scale) was 0.302. Figure 5 in Appendix A provides a comparison of random forest classifiers with up- and down-sampling, while Figure 6 shows ROC curves for a random forest classifier with up-sampling.

VI. Related Work

There is a large body of research on malware detection in contexts other than Android (e.g., [12], [13], [17], [33]–[39]). While our work was originally inspired by some of these approaches, we primarily focus in this section on more closely related work on Android malware detection. Ever since Android as a mobile computing platform has become popular, there is an increasing body of research on detecting malicious Android applications. We split Android malware detection work into static and dynamic analysis techniques.

Static Techniques. Static techniques are typically based on source code or binary analyses that search for malicious patterns (e.g., [6], [40]). For example, static approaches include analyzing permission requests for application installation [4], [41], [42], control flow [43], [44], signature-based detection [5], [6], and static taint-analysis [45].

Stowaway [46] is a tool that detects over-privilege requests during the application install time. Enck et al. [47] study popular applications by decompiling them back into their source code and then searching for unsafe coding security issues. Yang et al. [48] propose AppContext, a static program analysis approach to classify benign and malicious applications. AppContext classifies applications based on the contexts that trigger security-sensitive behaviors, by using machine learning techniques. It builds a call graph from an application binary and after different transformations it extracts the context factors via information flow analysis. It is then able to obtain the features for the machine learning algorithms from the extracted context. In the paper, 202 malicious and 633 benign applications from the Google Play store are analyzed. AppContext correctly identifies 192 malicious applications with an 87.7% accuracy.

Gascon et al. [9] propose to use function call graphs to detect malware. Once they extract function call graphs from Android applications, they apply a linear-time graph kernel in order to map call graphs to features. These features are given as input to SVMs to distinguish between benign and malicious applications. They conducted experiments on a total of 135,792 benign applications and 12,158 malware applications, detecting 89% of the malware with 1% of false positives.

Dynamic Techniques. Dynamic analysis techniques consist of running applications in a sandbox environment or on real devices in order to gather information about the application behavior. Dynamic taint analysis [49], [50] and behavior-based detection [16], [51] are examples of dynamic approaches. Our approach analyzes Android applications dynamically and captures their behavior based on the execution pattern of system calls. Some existing works follow similar approaches.

Dini et al. [51] propose a framework (MADAM) for Android malware detection which monitors applications at the kernel and user level. MADAM detects system calls at the kernel level and user activity/idleness at the user level to capture the application behavior. Then, it constructs feature vectors to apply machine learning techniques and classify those behaviors as benign or malicious. Their extremely preliminary and limited results, considering only 50 goodware and 2 malware applications, show 100% of an overall detection accuracy.

Crowdroid [16] is another behavior-based malware detector for Android that uses system calls and machine learning techniques. As opposed to our approach, Crowdroid collects

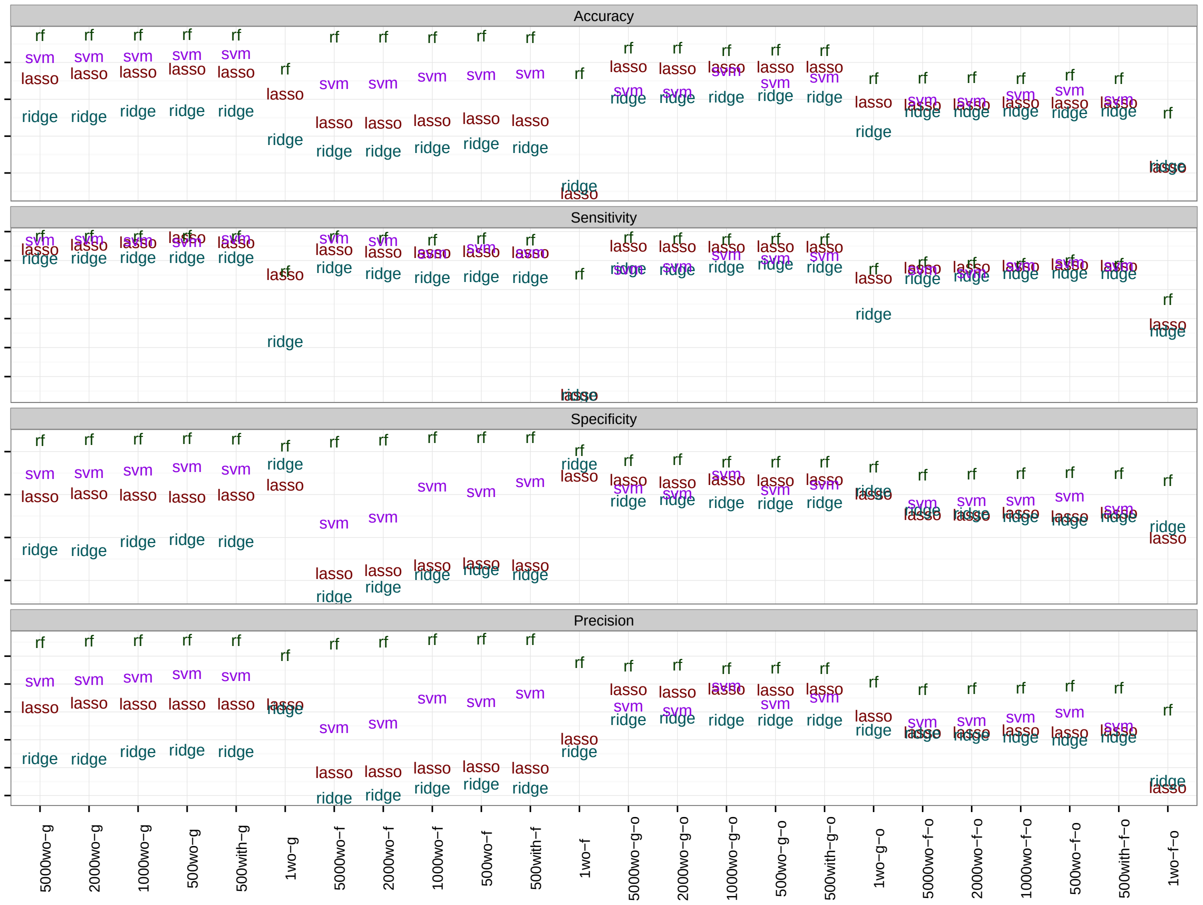

Fig. 3: Comparison of the quality of different classifiers through different quality measures (averaged on cross-validation folds). Labels on the x axis are written in the short form where *wo* stands for *without background*, *with* stands for *with background*, f stands for *freq*, g stands for *graph*, o at the end denotes that 0-1 matrices were used, and the numbers at the beginning represent numbers of generated events. 1-event experiments have a 2.3% smaller set of applications.

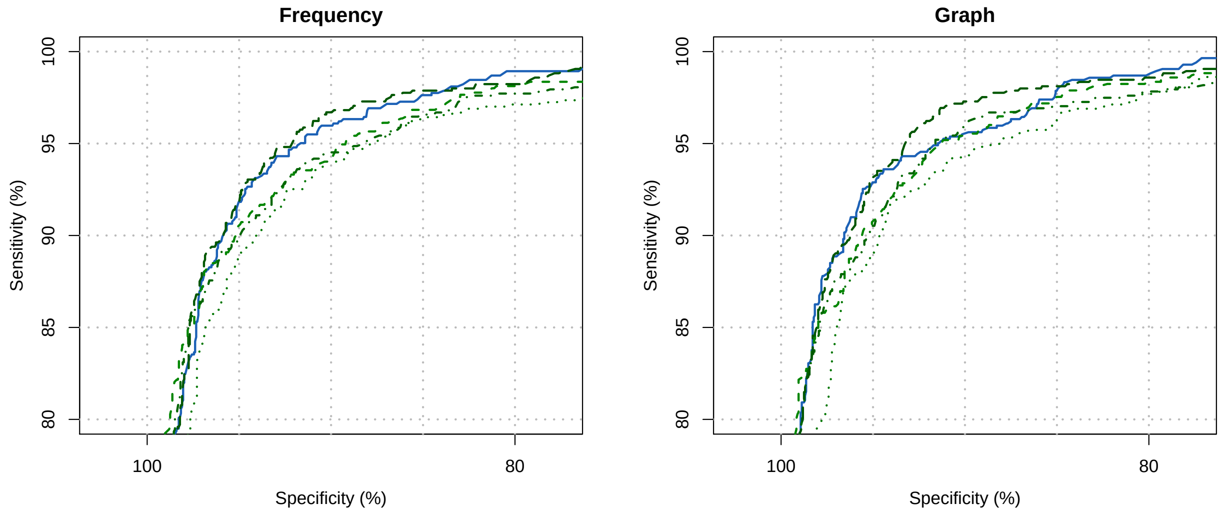

Fig. 4: ROC curves created from 5-fold cross-validation with random forest model on X_{freq}^{5000} and X_{graph}^{5000} matrices.

information about system calls through a community of users. A lightweight application, installed in the users' devices, monitors system calls (frequency) of running applications and sends them to a centralized server. The server produces feature vectors and applies a K-means clustering to classify the applications as malware or goodware. Crowdroid was evaluated on a limited number of goodware applications and only 2 real malware applications, obtaining detection accuracies of 100% for the first one and 85% for the second one.

VII. Threats to Validity

Application Crashes. Given that Monkey generates sequences of pseudo-random input events, it is to be expected that it can drive an application into a state that does not handle certain kinds of events, causing a crash. Depending on an experiment, we observed from 29% to 49% applications crash, which could bias our empirical results. However, it is important to note that the crash rate of goodware and malware applications is roughly the same. Therefore, application crashes do not bring in a classification bias.

Age of Applications. Our goodware data set comprises of applications downloaded in 2014, while our malware applications are from 2010 – 2012. Because the Android operating system's API evolved from 2010 to 2014, it could mean our approach learns differences between APIs, and not differences between benign and malicious behaviors. Unfortunately, we could not obtain older versions of applications from Google Play as it hosts only the most recent versions. In addition, to the best of our knowledge, a more recent malware data set does not exist. Hence, we manually downloaded 2010 – 2012 releases of 92 applications from F-Droid [52], an Android application repository offering multiple releases of free and open-source applications; we assumed the applications to be benign. We classified them using MALINE, and we got specificity of around 88%. Compared to the specificities from Figure 3, which were typically around 96%, this might indicate that MALINE performs API difference learning to some extent. However, a comparison with a much bigger set of the same applications across different releases would need to be performed to draw strong conclusions. This suggests that the difference in age of applications used in our experiments does not create a considerable bias.

Hidden Malicious Behavior. Malicious behavior may occasionally be hidden and triggered only under very specific circumstances. As our approach is based on random testing, we might miss such hard-to-reach behaviors, which could affect our ability to detect such application as malicious. Such malware is not common though, and ultimately we consistently get sensitivity of 87% and more using MALINE.

Detecting Emulation. As noted in previous work [35], [36], [53], malware could potentially detect it is running in an emulator, and alter its behavior accordingly. MALINE does not address this issue directly. However, an application trying to detect it is being executed in an emulator triggers numerous system calls, which likely leaves a specific signature that can be detected by MALINE. We consistently get sensitivity of 87% and more using MALINE. If we are to assume that all the remaining malware went undetected only due to its capability of detecting the emulator and consequently changing its behavior without leaving the system call signature, it is at most 13% of malware in our experiments that successfully disguise as goodware. Finally, Chen et al. [35] show that only less than 4% of malware in their experiments changes its behavior in a virtualized environment.

System Architecture and Native Code. While the majority of Android-powered devices are ARM-based, MALINE uses an x86-based Android emulator for performance reasons. Few Android applications — less than 5% according to Zhou. et al. [54] — contain native libraries typically compiled for multiple platforms, including x86, and hence they can be executed with MALINE. Nonetheless, the ARM and x86 system architectures have different system calls: with the x86-based and ARM-based emulator we observed applications utilizing 360 and 209 different system calls, respectively. Our initial implementation of MALINE was ARM-based, and switching to an x86-based implementation yielded roughly the same classification results in preliminary experiments, while it greatly improved performance.

Randomness in MALINE. We used only one seed value for Monkey's pseudo-random number generator; it is possible the outcome of our experiments would have been different if another seed value was used. However, as the seed value has to be used consistently within an experiment consisting of thousands of applications, it is highly unlikely the difference would be significant.

VIII. Conclusions and Future Work

We performed a preliminary feature selection exploration, but were not successful in obtaining consistent results. The reason could he a high dimensionality of the classification problem (15,000 features for our dependency representation) or a strong correlation between features. We left a more extensive feature selection exploration for future work. Another idea we want to explore in the future is combining several learning techniques to investigate ensemble learning. We already do use a form of ensemble learning in the random forest algorithm, but we are planning to look at combinations of algorithms too.

In this paper, we proposed a free and open-source reproducible research environment MALINE for dynamic-analysis-based malware detection in Android. We performed an extensive empirical evaluation of our novel system call encoding into a feature vector representation against a well-known frequency representation across several dimensions. The novel encoding showed better quality than the frequency representation. Our evaluation provides numerous insights into the structure of application executions, the impact of different machine learning techniques, and the type and size of inputs to dynamic analyses, serving as a guidance for future research.

Acknowledgments

We thank Raimondas Sasnauskas for helping us with the Android platform, and Geof Sawaya for providing feedback on an early draft of this paper. Thanks to Andreas Zeller and his group for providing us with a tool for bulk application downloading from Google Play, and the Flux Research Group of the University of Utah for the Emulab infrastructure. This work was supported in part by NSF CCF 1421678.

References

[1] "IDC: Smartphone OS market share 2014, 2013, 2012, and 2011." [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] "Third annual mobile threats report: March 2012 through March 2013," Juniper Networks Mobile Threat Center (MTC). [Online]. Available: http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf

[3] "Riskiq reports malicious mobile apps in google play have spiked nearly 400 percent," http://www.riskiq.com/company/press-releases/riskiq-reports-malicious-mobile-apps-google-play-have-spiked-nearly-400, 2014.

[4] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Security and Privacy in Communication Networks*, 2013, pp. 86–103.

[5] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: scalable and accurate zero-day Android malware detection," in *10th International Conference on Mobile Systems, Applications, and Services*, 2012, pp. 281–294.

[6] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 576–587.

[7] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *Computer Security — ESORICS*, 2012, vol. 7459, pp. 37–54.

[8] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013, vol. 7591, pp. 62–81.

[9] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *ACM Workshop on Artificial Intelligence and Security*, 2013, pp. 45–54.

[10] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android Application Sandbox system for suspicious software detection," in *5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010, pp. 55–62.

[11] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors," *European Workshop on System Security*, 2013.

[12] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "AccessMiner: Using System-centric Models for Malware Protection," in *17th ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 399–412.

[13] S. Palahan, D. Babić, S. Chaudhuri, and D. Kifer, "Extraction of Statistically Significant Malware Behaviors," in *29th Computer Security Applications Conference (ACSAC)*, 2013.

[14] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, 2nd ed. Springer, 2009.

[15] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*, ser. Springer Texts in Statistics. Springer, 2013.

[16] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for Android," in *1st ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, 2011, pp. 15–26.

[17] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *IEEE Symposium on Security and Privacy (SP)*, 2010, pp. 45–60.

[18] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[19] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[20] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society, Series B*, vol. 58, pp. 267–288, 1994.

[21] A. Tikhonov and V. Arsenin, *Solutions of ill-posed problems*, ser. Scripta series in mathematics, 1977.

[22] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.

[23] M. Kuhn and K. Johnson, *Applied Predictive Modeling*. Springer, 2013. [Online]. Available: http://books.google.hr/books?id=xYRDAAAAQBAJ

[24] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[25] "UI/application exerciser monkey." [Online]. Available: http://developer.android.com/tools/help/monkey.html

[26] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 27:1–27:27, 2011.

[27] "The R project for statistical computing." [Online]. Available: http://www.r-project.org

[28] "Comprehensive R Archive Network (CRAN)." [Online]. Available: http://CRAN.R-project.org

[29] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 255–270, 2002.

[30] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[31] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy (SP)*, 2012, pp. 95–109.

[32] M. Ojala and G. C. Garriga, "Permutation tests for studying classifier performance," *The Journal of Machine Learning Research (JMLR)*, vol. 11, pp. 1833–1863, 2010.

[33] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[34] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: Protecting sensitive data leaks using application-level taint tracking," *SIGOPS Operating System Review*, vol. 45, no. 1, pp. 142–154, 2011.

[35] X. Chen, J. Andersen, Z. 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 With FTCS and DCC, 2008. DSN 2008*, 2008, pp. 177–186.

[36] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect CPU emulators," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[37] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *USENIX Security Symposium*, 2009, pp. 351–366.

[38] T. Singh, F. D. Troia, V. A. Corrado, T. H. Austin, and M. Stamp, "Support vector machines and malware detection," *Journal of Computer Virology and Hacking Techniques*, 2015.

[39] R. K. Jidigam, T. H. Austin, and M. Stamp, "Singular value decomposition and metamorphic detection," *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 4, 2014.

[40] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014, pp. 1329–1341.

[41] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *36th International Conference on Software Engineering (ICSE)*, 2014, pp. 1025–1035.

[42] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *8th Symposium on Usable Privacy and Security (SOUPS)*, 2012, pp. 3:1–3:14.

[43] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. Van Horn, "Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation," in *3rd ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, 2013, pp. 21–32.

[44] S. Liang, W. Sun, and M. Might, "Fast flow analysis with Gödel hashes," in *14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2014, pp. 225–234.

[45] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps,” in *35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 259–269.

[46] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, “Android permissions demystified,” in *18th ACM Conference on Computer and Communications Security (CCS)*, 2011, pp. 627–638.

[47] E. William, D. Octeau, P. McDaniel, and S. Chaudhuri, “A study of Android application security,” in *USENIX Security Symposium*, 2011.

[48] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and E. William, “Appcontext: Differentiating malicious and benign mobile app behavior under contexts,” in *International Conference on Software Engineering (ICSE)*, 2015, to appear.

[49] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” in *9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 1–6.

[50] L. K. Yan and H. Yin, “Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis,” in *21st USENIX Security Symposium*, 2012.

[51] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, “MADAM: A multi-level anomaly detector for Android malware,” in *Computer Network Security (CNS)*, 2012, pp. 240–253.

[52] “F-droid, free and open source android app repository.” [Online]. Available: https://f-droid.org/

[53] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, “Morpheus: automatically generating heuristics to detect Android emulators,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 216–225.

[54] 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 *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

APPENDIX

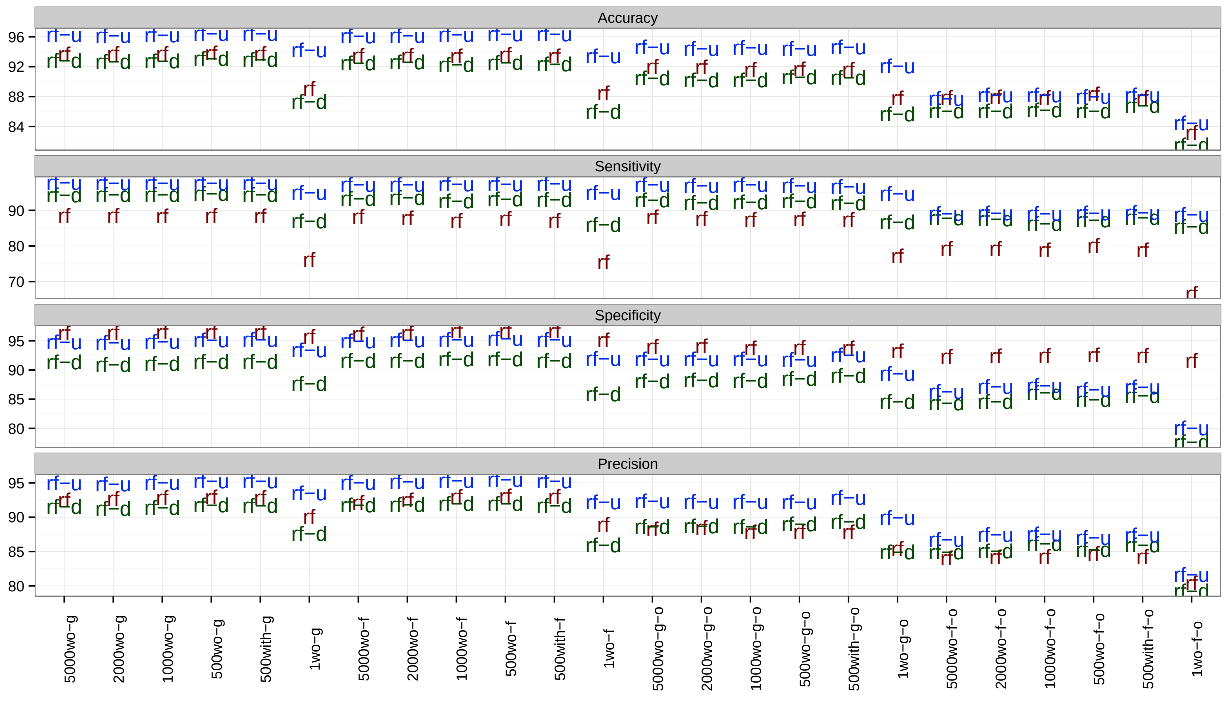

Fig. 5: Comparison of the quality of random forest classifier with up- and down-sampling on different initial data sets (averaged on 5 cross-validation folds). Labels on the x axis are written in the short form where *wo* stands for *without background*, *with* stands for *with background*, *f* stands for *freq*, *g* stands for *graph*, *o* at the end denotes that 0-1 matrices were used, and the numbers at the beginning represent number of events used. In the names of classifiers *-u* denotes up-sampling while *-d* denotes down-sampling.

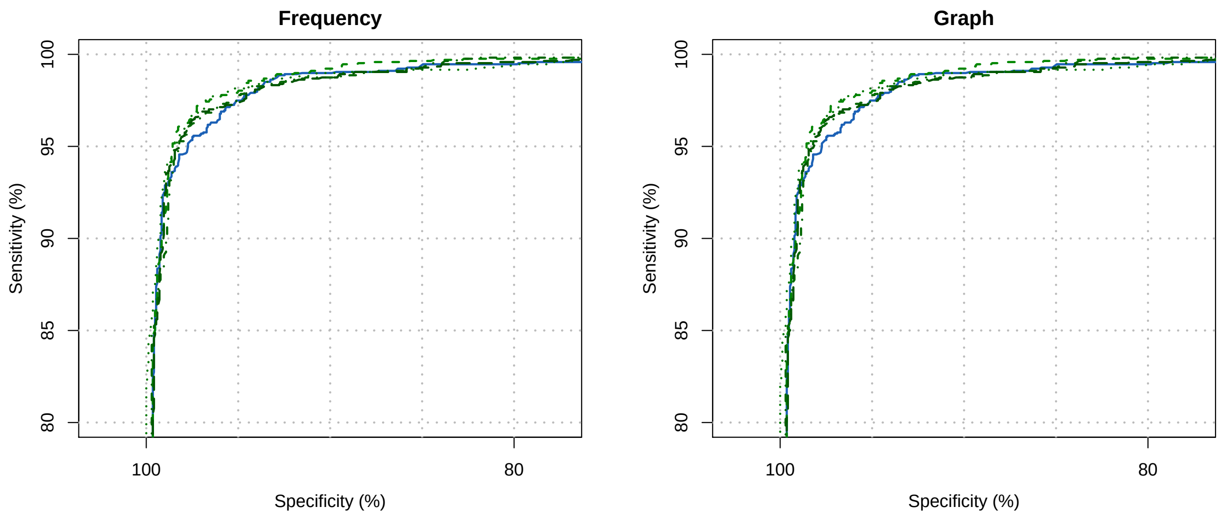

Fig. 6: ROC curves created from 5-fold cross-validation with random forest model on X_{freq}^{5000} and X_{graph}^{5000} feature matrices with up-sampling.

	Accuracy				Precision				Sensitivity				Specificity			
Name	lasso	ridge	rf	svm	lasso	ridge	rf	svm	lasso	ridge	rf	svm	lasso	ridge	rf	svm
5000wo-g	82.69	87.84	93.71	90.69	71.70	80.85	92.51	85.63	80.80	83.99	88.63	87.15	83.66	89.81	96.32	92.49
2000wo-g	82.69	88.55	93.76	90.86	71.60	81.63	92.73	85.84	81.06	85.46	88.54	87.52	83.53	90.14	96.44	92.58
1000wo-g	83.43	88.75	93.78	90.91	72.92	81.50	92.87	86.27	81.25	86.41	88.44	87.03	84.54	89.96	96.51	92.89
500wo-g	83.55	89.17	93.85	91.10	73.22	81.47	92.94	86.95	81.08	88.06	88.61	86.75	84.80	89.74	96.55	93.32
500with-g	83.43	88.75	93.78	91.23	72.92	81.50	92.87	86.58	81.25	86.41	88.44	87.74	84.54	89.96	96.51	93.02
1wo-g	79.57	85.79	89.13	77.56	80.69	81.38	90.13	67.38	52.25	75.33	76.28	65.56	93.59	91.17	95.72	83.69
5000wo-f	78.02	81.91	93.52	87.15	64.58	69.22	92.20	77.30	77.74	83.98	88.35	87.95	78.15	80.85	96.16	86.72
2000wo-f	78.02	81.80	93.48	87.27	65.11	69.35	92.56	78.03	75.61	82.99	87.84	86.99	79.25	81.19	96.38	87.43
1000wo-f	78.52	82.18	93.44	88.21	66.38	69.98	93.06	82.53	74.26	82.96	87.16	82.71	80.72	81.77	96.67	91.03
500wo-f	79.05	82.40	93.66	88.39	67.15	70.30	93.14	81.94	74.66	83.22	87.74	84.57	81.29	81.99	96.69	90.38
500with-f	78.52	82.18	93.44	88.61	66.38	69.98	93.06	83.41	74.26	82.96	87.16	82.88	80.72	81.77	96.67	91.56
1wo-f	73.29	72.25	88.52	83.43	72.94	75.23	88.92	80.17	33.72	33.81	75.56	68.01	93.59	92.21	95.17	91.35
5000wo-g-o	85.24	89.51	92.03	86.10	78.71	84.08	88.33	88.76	77.38	85.18	88.14	90.46	89.27	91.73	94.02	77.60
2000wo-g-o	85.28	89.23	91.97	85.94	78.96	83.58	88.50	89.00	77.10	84.88	87.75	89.83	89.48	91.46	94.14	78.32
1000wo-g-o	85.32	89.42	91.64	88.92	78.58	84.12	87.88	91.06	77.88	84.75	87.39	92.30	89.13	91.80	93.82	82.31
500wo-g-o	85.58	89.39	91.72	87.34	78.62	83.97	87.93	90.48	78.89	84.92	87.61	90.34	89.01	91.68	93.83	81.49
500with-g-o	85.32	89.42	91.64	88.05	78.58	84.12	87.88	90.94	77.88	84.75	87.39	90.99	89.13	91.80	93.82	82.31
1wo-g-o	80.69	84.66	87.83	79.42	76.81	79.32	85.44	73.34	61.66	74.05	77.25	61.85	90.46	90.10	93.26	88.42
5000wo-f-o	83.35	84.27	87.92	84.79	76.24	76.40	84.07	88.40	73.85	77.50	79.39	88.63	88.21	87.73	92.29	77.28
2000wo-f-o	83.40	84.33	87.98	84.67	75.89	76.37	84.26	88.27	74.71	77.85	79.34	88.58	87.84	87.65	92.40	77.03
1000wo-f-o	83.48	84.61	87.89	85.60	75.57	76.75	84.33	89.07	75.70	78.26	78.92	89.17	87.47	87.87	92.49	78.64
500wo-f-o	83.25	84.52	88.36	86.28	75.01	76.32	84.71	89.58	75.83	78.73	80.13	89.68	87.05	87.48	92.58	79.66
500with-f-o	83.48	84.61	87.89	84.88	75.57	76.75	84.33	88.99	75.70	78.26	78.92	88.03	87.47	87.87	92.49	78.72
1wo-f-o	75.96	75.87	83.20	75.55	67.63	66.51	80.42	66.84	55.77	58.09	66.68	55.55	86.32	85.00	91.67	85.80

TABLE III: Comparison of the quality of different classifiers through different quality measures corresponding to Figure 3.