Drebin: Effective and Explainable Detection of Android Malware in Your Pocket

Daniel Arp[*], Michael Spreitzenbarth[†], Malte Hübner[*], Hugo Gascon[*], and Konrad Rieck[*]
[*]University of Göttingen, Germany
Email: firstname.lastname@cs.uni-goettingen.de
[†]Siemens AG, Munich, Germany
Email: michael.spreitzenbarth@siemens.com

***Abstract*—Malicious applications pose a threat to the security of the Android platform. The growing amount and diversity of these applications render conventional defenses largely ineffective and thus Android smartphones often remain unprotected from novel malware. In this paper, we propose DREBIN, a lightweight method for detection of Android malware that enables identifying malicious applications directly on the smartphone. As the limited resources impede monitoring applications at run-time, DREBIN performs a broad static analysis, gathering as many features of an application as possible. These features are embedded in a joint vector space, such that typical patterns indicative for malware can be automatically identified and used for explaining the decisions of our method. In an evaluation with 123,453 applications and 5,560 malware samples DREBIN outperforms several related approaches and detects 94% of the malware with few false alarms, where the explanations provided for each detection reveal relevant properties of the detected malware. On five popular smartphones, the method requires 10 seconds for an analysis on average, rendering it suitable for checking downloaded applications directly on the device.**

I. INTRODUCTION

Android is one of the most popular platforms for smartphones today. With several hundred thousands of applications in different markets, it provides a wealth of functionality to its users. Unfortunately, smartphones running Android are increasingly targeted by attackers and infected with malicious software. In contrast to other platforms, Android allows for installing applications from unverified sources, such as third-party markets, which makes bundling and distributing applications with malware easy for attackers. According to a recent study over 55,000 malicious applications and 119 new malware families have been discovered in 2012 alone [18]. It is evident that there is a need for stopping the proliferation of malware on Android markets and smartphones.

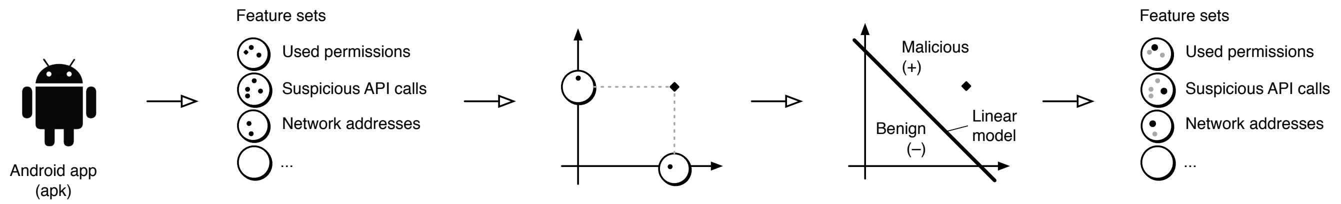
The Android platform provides several security measures that harden the installation of malware, most notably the Android permission system. To perform certain tasks on the device, such as sending a SMS message, each application has to explicitly request permission from the user during the installation. However, many users tend to blindly grant permissions to unknown applications and thereby undermine the purpose of the permission system. As a consequence, malicious applications are hardly constrained by the Android permission system in practice.

A large body of research has thus studied methods for analyzing and detecting Android malware prior to their installation. These methods can be roughly categorized into approaches using static and dynamic analysis. For example, TaintDroid [11], DroidRanger [40] and DroidScope [37] are methods that can monitor the behavior of applications at run-time. Although very effective in identifying malicious activity, run-time monitoring suffers from a significant overhead and can not be directly applied on mobile devices. By contrast, static analysis methods, such as Kirin [13], Stowaway [15] and RiskRanker [21], usually induce only a small run-time overhead. While these approaches are efficient and scalable, they mainly build on manually crafted detection patterns which are often not available for new malware instances. Moreover, most of these methods do not provide explanations for their decisions and are thus opaque to the practitioner.

In this paper, we present DREBIN, a lightweight method for detection of Android malware that infers detection patterns automatically and enables identifying malware directly on the smartphone. DREBIN performs a broad static analysis, gathering as many features from an application's code and manifest as possible. These features are organized in sets of strings (such as permissions, API calls and network addresses) and embedded in a joint vector space. As an example, an application sending premium SMS messages is cast to a specific region in the vector space associated with the corresponding permissions, intents and API calls. This geometric representation enables DREBIN to identify combinations and patterns of features indicative for malware automatically using machine learning techniques. For each detected application the respective patterns can be extracted, mapped to meaningful descriptions and then provided to the user as explanation for the detection. Aside from detection, DREBIN can thus also provide insights into identified malware samples.

Experiments with 123,453 applications from different markets and 5,560 recent malware samples demonstrate the efficacy of our method: DREBIN outperforms related approaches [13, 26, 33] as well as 9 out of 10 popular anti-virus scanners. The method detects 94% of the malware samples with a false-positive rate of 1%, corresponding to one false alarm in 100 installed applications. On average the analysis of an application requires less than a second on a regular computer and 10 seconds on popular smartphone models. To the best of our

NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23247



(a) Broad static analysis (b) Embedding in vector space (c) Learning-based detection (d) Explanation

Fig. 1: Schematic depiction of the analysis steps performed by DREBIN.

knowledge, DREBIN is the first method which provides effective and explainable detection of Android malware directly on smartphone devices.

In summary, we make the following contributions to the detection of Android malware in this paper:

- *Effective detection.* We introduce a method combining static analysis and machine learning that is capable of identifying Android malware with high accuracy and few false alarms, independent of manually crafted detection patterns.
- *Explainable results.* The proposed method provides an explainable detection. Patterns of features indicative for a detected malware instance can be traced back from the vector space and provide insights into the detection process.
- *Lightweight analysis.* For efficiency we apply linear-time analysis and learning techniques that enable detecting malware on the smartphone as well as analyzing large sets of applications in reasonable time.

We need to note here that DREBIN builds on concepts of static analysis and thus cannot rule out the presence of obfuscated or dynamically loaded malware on mobile devices. We specifically discuss this limitation of our approach in Section IV. Due to the broad analysis of features however, our method raises the bar for attackers to infect smartphones with malicious applications and strengthens the security of the Android platform, as demonstrated in our evaluation.

The rest of this paper is organized as follows: DREBIN and its detection methodology are introduced in Section II. Experiments and a comparison with related approaches are presented in Section III. Limitations and related work are discussed in Section IV and Section V, respectively. Section VI concludes the paper.

II. METHODOLOGY

To detect malicious software on a smartphone, DREBIN requires a comprehensive yet lightweight representation of applications that enables determining typical indications of malicious activity. To this end, our method employs a broad static analysis that extracts feature sets from different sources and analyzes these in an expressive vector space. This process is illustrated in Figure 1 and outlined in the following:

a) *Broad static analysis.* In the first step, DREBIN statically inspects a given Android application and extracts different feature sets from the application's manifest and dex code (Section II-A).

b) *Embedding in vector space.* The extracted feature sets are then mapped to a joint vector space, where patterns and combinations of the features can be analyzed geometrically (Section II-B).

c) *Learning-based detection.* The embedding of the featue sets enables us to identify malware using efficient techniques of machine learning, such as linear Support Vector Machines (Section II-C).

d) *Explanation.* In the last step, features contributing to the detection of a malicious application are identified and presented to the user for explaining the detection process (Section II-D).

In the following sections, we discuss these four steps in more detail and provide necessary technical background of the analysis.

A. Static Analysis of Applications

As the first step, DREBIN performs a lightweight static analysis of a given Android application. Although apparently straightforward, the static extraction of features needs to run in a constrained environment and complete in a timely manner. If the analysis takes too long, the user might skip the ongoing process and refuse the overall method. Accordingly, it becomes essential to select features which can be extracted efficiently.

We thus focus on the manifest and the disassembled dex code of the application, which both can be obtained by a linear sweep over the application's content. To allow for a generic and extensible analysis, we represent all extracted features as sets of strings, such as permissions, intents and API calls. In particular, we extract the following 8 sets of strings.

1) Feature sets from the manifest: Every application developed for Android must include a manifest file called *AndroidManifest.xml* which provides data supporting the installation and later execution of the application. The information stored in this file can be efficiently retrieved on the device using the *Android Asset Packaging Tool* that enables us to extract the following sets:

S_1 *Hardware components:* This first feature set contains requested hardware components. If an application requests access to the camera, touchscreen or the GPS module of the smartphone, these features need to be declared in the manifest file. Requesting access to specific hardware has clearly security implications, as the use of certain combinations of hardware often reflects harmful behavior. An application which has access to GPS and network modules is, for instance, able to collect location data and send it to an attacker over the network.

S_2 *Requested permissions:* One of the most important security mechanisms introduced in Android is the permission system. Permissions are actively granted by the user at installation time and allow an application to access security-relevant resources. As shown by previous work [13, 33], malicious software tends to request certain permissions more often than innocuous applications. For example, a great percentage of current malware sends premium SMS messages and thus requests the `SEND_SMS` permission. We thus gather all permissions listed in the manifest in a feature set.

S_3 *App components:* There exist four different types of components in an application, each defining different interfaces to the system: *activities*, *services*, *content providers* and *broadcast receivers*. Every application can declare several components of each type in the manifest. The names of these components are also collected in a feature set, as the names may help to identify well-known components of malware. For example, several variants of the so-called DroidKungFu family share the name of particular services [see 24].

S_4 *Filtered intents:* Inter-process and intra-process communication on Android is mainly performed through *intents*: passive data structures exchanged as asynchronous messages and allowing information about events to be shared between different components and applications. We collect all intents listed in the manifest as another feature set, as malware often listens to specific intents. A typical example of an intent message involved in malware is `BOOT_COMPLETED`, which is used to trigger malicious activity directly after rebooting the smartphone.

2) Feature sets from disassembled code: Android applications are developed in Java and compiled into optimized bytecode for the Dalvik virtual machine. This bytecode can be efficiently disassembled and provides DREBIN with information about API calls and data used in an application. To achieve a low run-time, we implement a lightweight disassembler based on the dex libraries of the Android platform that can output all API calls and strings contained in an application. We use this information to construct the following feature sets.

S_5 *Restricted API calls:* The Android permission system restricts access to a series of critical API calls. Our method searches for the occurrence of these calls in the disassembled code in order to gain a deeper understanding of the functionality of an application. A particular case, revealing malicious behavior, is the use of restricted API calls for which the required permissions have not been requested. This may indicate that the malware is using root exploits in order to surpass the limitations imposed by the Android platform.

S_6 *Used permissions:* The complete set of calls extracted in S_5 is used as the ground for determining the subset of permissions that are both requested and actually used. For this purpose, we implement the method introduced by Felt et al. [15] to match API calls and permissions. In contrast to S_5, this feature set provides a more general view on the behavior of an application as multiple API calls can be protected by a single permission (e.g., `sendMultipartTextMessage()` and `sendTextMessage()` both require that the `SEND_SMS` permission is granted to an application).

S_7 *Suspicious API calls:* Certain API calls allow access to sensitive data or resources of the smartphone and are frequently found in malware samples. As these calls can specially lead to malicious behavior, they are extracted and gathered in a separated feature set. In particular, we collect the following types of API calls:

- API calls for accessing sensitive data, such as `getDeviceId()` and `getSubscriberId()`
- API calls for network communication, for example `execHttpRequest()` and `setWifiEnabled()`
- API calls for sending and receiving SMS messages, such as `sendTextMessage()`
- API calls for execution of external commands like `Runtime.exec()`
- API calls frequently used for obfuscation, such as `Cipher.getInstance()`

S_8 *Network addresses:* Malware regularly establishes network connections to retrieve commands or exfiltrate data collected from the device. Therefore, all IP addresses, hostnames and URLs found in the disassembled code are included in the last set of features. Some of these addresses might be involved in botnets and thus present in several malware samples, which can help to improve the learning of detection patterns.

B. Embedding in Vector Space

Malicious activity is usually reflected in specific patterns and combinations of the extracted features. For example, a malware sending premium SMS messages might contain the permission `SEND_SMS` in set S_2, and the hardware component `android.hardware.telephony` in set S_1. Ideally, we would like to formulate Boolean expressions that capture these dependencies between features and return true if a malware is detected. However, inferring Boolean expressions from real-world data is a hard problem and difficult to solve efficiently.

As a remedy, we aim at capturing the dependencies between features using concepts from machine learning. As most learning methods operate on numerical vectors, we first need to map the extracted feature sets to a vector space. To this end, we define a joint set S that comprises all observable strings contained in the 8 feature sets

$$S := S_1 \cup S_2 \cup \cdots \cup S_8.$$

We ensure that elements of different sets do not collide by adding a unique prefix to all strings in each feature set. In our evaluation the set S contains roughly 545,000 different features (see Section III).

Using the set S, we define an $|S|$-dimensional vector space, where each dimension is either 0 or 1. An application x is mapped to this space by constructing a vector $\varphi(x)$, such that for each feature s extracted from x the respective dimension is set to 1 and all other dimensions are 0. Formally, this map φ can be defined for a set of applications X as follows

$$\varphi : X \to \{0,1\}^{|S|}, \quad \varphi(x) \mapsto \big(I(x,s)\big)_{s\in S}$$

where the indicator function $I(x,s)$ is simply defined as

$$I(x,s) = \begin{cases} 1 & \text{if the application } x \text{ contains feature } s \\ 0 & \text{otherwise.} \end{cases}$$

Applications sharing similar features lie close to each other in this representation, whereas applications with mainly different features are separated by large distances. Moreover, directions in this space can be used to describe combinations of features and ultimately enable us to learn explainable detection models.

Let us, as an example, consider a malicious application that sends premium SMS messages and thus needs to request certain permissions and hardware components. A corresponding vector $\varphi(x)$ for this application looks like this

$$\varphi(x) \mapsto \begin{pmatrix} \cdots \\ 0 \\ 1 \\ \cdots \\ 1 \\ 0 \\ \cdots \end{pmatrix} \begin{array}{l} \cdots \\ \left.\begin{array}{l} \texttt{android.hardware.wifi} \\ \texttt{android.hardware.telephony} \end{array}\right\} S_1 \\ \cdots \\ \left.\begin{array}{l} \texttt{SEND\_SMS} \\ \texttt{DELETE\_PACKAGES} \end{array}\right\} S_2 \\ \cdots \end{array}$$

At a first glance, the map φ seems inappropriate for the lightweight analysis of applications, as it embeds data into a high-dimensional vector space. Fortunately, the number of features extracted from an application is linear in its size. That is, an application x containing m bytes of code and data contains at most m feature strings. As a consequence, only m dimensions are non-zero in the vector $\varphi(x)$—irrespective of the dimension of the vector space. It thus suffices to only store the features extracted from an application for sparsely representing the vector $\varphi(x)$, for example, using hash tables [6] or Bloom filters [3].

C. Learning-based Detection

In the third step, we apply machine learning techniques for automatically learning a separation between malicious and benign applications. The application of machine learning spares us from manually constructing detection rules for the extracted features.

While several learning methods can be applied to learn a separation between two classes, only few methods are capable of producing an efficient and explainable detection model. We consider linear *Support Vector Machines* (SVM) [8, 14] for this task. Given vectors of two classes as training data, a linear SVM determines a hyperplane that separates both classes with maximal margin (see Fig. 2). One of these classes is associated with malware, whereas the other class corresponds to benign applications. An unknown application is classified by mapping it to the vector space and determing whether it falls on the malicious $(+)$ or benign $(-)$ side of the hyperplane.

Formally, the detection model of a linear SVM simply corresponds to a vector $w \in \mathbb{R}^{|S|}$ specifying the direction of

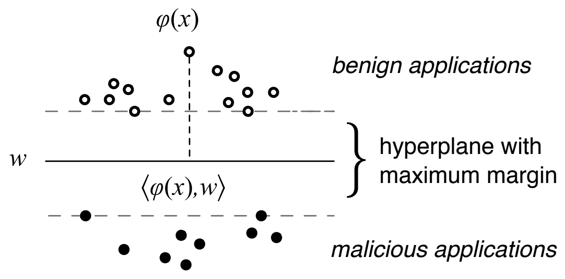


Fig. 2: Schematic depiction of an SVM

the hyperplane, where the corresponding detection function f is given by

$$f(x) = \langle \varphi(x), w \rangle = \sum_{s\in S} I(x,s) \cdot w_s$$

and returns the orientation of $\varphi(x)$ with respect to w. That is, $f(x) > t$ indicates malicious activity, while $f(x) \le t$ corresponds to benign applications for a given threshold t.

To compute the function f efficiently, we again exploit the sparse representation of the map φ. Given an application x, we know that only features extracted from x have non-zero entries in $\varphi(x)$. All other dimensions are zero and do not contribute to the computation of $f(x)$. Hence, we can simplify the detection function f as follows

$$f(x) = \sum_{s\in S} I(x,s) \cdot w_s = \sum_{s\in x} w_s.$$

Instead of an involved learning model, we finally arrive at simple sum that can be efficiently computed by just adding the weight w_s for each feature s in an application x. This formulation enables us to apply a learned detection model on a smartphone and also allows us to explain results obtained by the Support Vector Machine.

Offline learning: In our implementation we do not learn a detection model on the smartphone. Instead, we train the Support Vector Machine offline on a dedicated system and only transfer the learned model w to the smartphone for detecting malicious applications.

D. Explanation

In practice, a detection system must not only indicate malicious activity, but also provide explanations for its detection results. It is a common shortcoming of learning-based approaches that they are black-box methods [34]. In the case of DREBIN, we address this problem and extend our learning-based detection, such that it can identify features of an application that contribute to a detection. Moreover, an explainable detection may also help researchers to inspect patterns in malware and gain a deeper understanding of its functionality.

By virtue of the simple detection function of the linear SVM, we are able to determine the contribution of each single feature s to the function $f(x)$. During the computation of $f(x)$, we just need to store the largest k weights w_s shifting the application to the malicious side of the hyperplane. Since each weight w_s is assigned to a certain feature s, it is then possible to explain why an application has been classified as malicious or not. This approach can be efficiently realized by maintaining the k largest weights w_s in a heap during the computation of the function $f(x)$ [6].

After extracting the top k features by their weights, DREBIN automatically constructs sentences which describe the functionality underlying these features. To achieve this goal we design sentence templates for each feature set which can be completed using the respective feature. Table I lists these templates. For features frequently observed in malware, such as the permission `SEND_SMS`, we provide individual descriptions.

Feature set	Explanation
S_1 Hardware features	App uses %s feature %s.
S_2 Requested permissions	App requests permission to access %s.
S_3 App components	App contains suspicious component %s.
S_4 Filtered intents	Action is triggered by %s.
S_5 Restricted API calls	App calls function %s to access %s.
S_6 Used permissions	App uses permissions %s to access %s.
S_7 Suspicious API calls	App uses suspicious API call %s.
S_8 Network addresses	Communication with host %s.

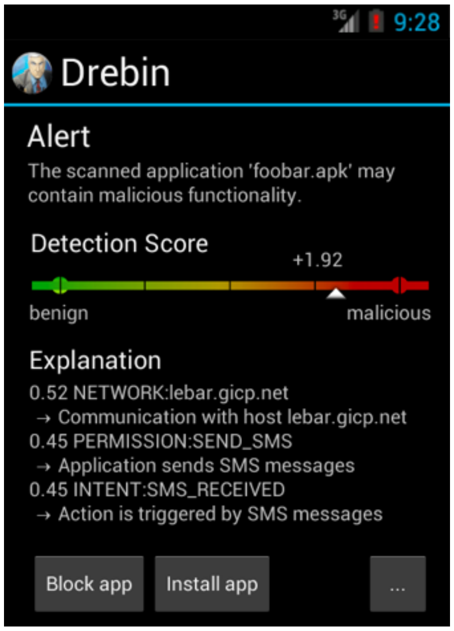
TABLE I: Templates for explanation.

For most of the feature sets, the construction of sentences from the templates in Table I is straightforward. For example, for the hardware features we make use of their naming scheme to construct meaningful sentences. If an application for instance uses the `android.hardware.camera` feature, DREBIN presents the sentence *"App uses hardware feature camera."* to the user.

Similarly, we provide explanations for requested and used permissions. The explanation for a permissions can be derived from the Android documentation which provides proper descriptions—at least for all system permissions. We slightly modify these descriptions in order to present meaningful explanations to the user. However, due to the fact that application developers are able to define custom permissions we also provide a generic sentence which is presented to the user if no proper description exists. We follow the same approach for the restricted API calls that build on the use of certain permissions. For all other feature sets the templates are directly filled with either the feature's name or a corresponding place holder.

An example of an explanation generated by DREBIN is shown in Figure 3. The presented sample belongs to the GoldDream family. DREBIN correctly identifies that the malware communicates with an external server and sends SMS messages. The application requests 16 permissions during the installation process. Many users ignore such long lists of permissions and thereby fall victim to this type of malware. In contrast to the conventional permission-based approach DREBIN draws the user's attention directly to relevant aspects indicating malicious activity. Furthermore, DREBIN presents a score to the user which tells him how confident the decision is. As a result, the user is able to decide whether the presented functionality matches his expectation or not.

In addition to the benefit for the user, the generated explanations can also help researchers to discover relevant patterns in common malware families. We discuss this aspect in more detail in the following section.



Alert

The scanned application 'foobar.apk' may contain malicious functionality.

Explanation

0.52 NETWORK:lebar.gicp.net
→ Communication with host lebar.gicp.net
0.45 PERMISSION:SEND_SMS
→ Application sends SMS messages
0.45 INTENT:SMS_RECEIVED
→ Action is triggered by SMS messages

Fig. 3: Result for a member of the GoldDream family.

III. EVALUATION

After presenting DREBIN in detail, we now proceed to an empirical evaluation of its efficacy. In particular, we conduct the following three experiments:

1) *Detection performance.* First, we evaluate the detection performance of DREBIN on a dataset of 5,560 malware samples and 123,453 benign applications. We compare its performance against related approaches and anti-virus scanners (Section III-B).
2) *Explainability.* In the second experiment, we analyze the explanations provided by DREBIN in detail for different malware families and verify whether they relate to actual characteristics of the malware (Section III-C).
3) *Run-time performance.* Finally, we evaluate the run-time performance of DREBIN. For this experiment we conduct different run-time measurements using five common smartphone models as well as a regular desktop computer (Section III-D).

A. Data sets

For all experiments, we consider a dataset of real Android applications and real malware. In particular, we have acquired an initial dataset of 131,611 applications comprising benign as well as malicious software. The samples have been collected in the period from August 2010 to October 2012. In detail, the dataset contains 96,150 applications from the GooglePlay Store, 19,545 applications from different alternative Chinese Markets, 2,810 applications from alternative Russian Markets and 13,106 samples from other sources, such as Android websites, malware forums and security blogs. Additionally, the dataset includes all samples from the *Android Malware Genome Project* [39].

To determine malicious and benign applications, we send each sample to the VirusTotal service and inspect the output

	DREBIN	AV1	AV2	AV3	AV4	AV5	AV6	AV7	AV8	AV9	AV10
Full dataset	93.90%	96.41%	93.71%	84.66%	84.54%	78.38%	64.16%	48.50%	48.34%	9.84%	3.99%
Malgenome	95.90%	98.63%	98.90%	98.28%	98.07%	98.66%	96.49%	94.67%	84.23%	23.68%	1.12%

TABLE II: Detection rates of DREBIN and anti-virus scanners.

of ten common anti-virus scanners (AntiVir, AVG, BitDefender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda, Sophos). We flag all applications as malicious that are detected by at least two of the scanners. This procedure ensures that our data is (almost) correctly split into benign and malicious samples—even if one of the ten scanners falsely labels a benign application as malicious.

Finally, we remove samples labeled as adware from our dataset, as this type of software is in a twilight zone between malware and benign functionality. The final dataset contains 123,453 benign applications and 5,560 malware samples. To the best of our knowledge, this is one of the largest malware datasets that has been used to evaluate a malware detection method on Android.

An overview of the top 20 malware families in our dataset is provided in Table 4(c) including several families that are currently actively distributed in application markets. Note that only the top 20 families are shown and our dataset contains 1,048 further malicious samples.

B. Detection Performance

In our first experiment, we evaluate the detection performance of DREBIN and related static detection approaches. For this experiment, we randomly split the dataset into a known partition (66%) and an unknown partition (33%). The detection model and respective parameters of DREBIN are determined on the known partition, whereas the unknown partition is only used for measuring the final detection performance. We repeat this procedure 10 times and average results. The partitioning ensures that reported results only refer to malicious applications unknown during the learning phase of DREBIN. For the related approaches, such as Kirin [13] and RCP [33] this experimental procedure differs slightly, since not all methods require a separate training step.

1) Comparison with related approaches: We first compare the performance of DREBIN against related static approaches for detection of Android malware. In particular, we consider the methods Kirin [13], RCP [33] and the approach by Peng et al. [26], where we implement the latter using an SVM instead of a Naive Bayes classifier. The results of this experiments are shown in Figure 4(a) as ROC curve, that is, the detection rate (true-positive rate) is plotted against the false-positive rate for different thresholds of the detection methods.

DREBIN significantly outperforms the other approaches and detects 94% of the malware samples at a false-positive rate of 1%, corresponding to one false alarm when installing 100 applications. The other approaches provide a detection rate between 10%–50% at this false-positive rate. As Kirin and RCP both consider only a subset of the requested permissions, they have obvious limitations in detecting malicious applications. Even the method by Peng et al. which considers all permissions is unable to detect malware with sufficient accuracy in this experiment. The good performance of DREBIN results from the different feature sets that are used to model malicious activity. These sets include the requested permissions but also contain other relevant characteristics of applications, such as suspicious API calls, filtered intents and network addresses.

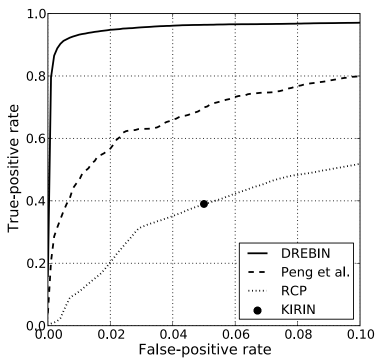
2) Comparison with AV scanners: Although DREBIN shows a better performance compared to related approaches, in the end it has to compete with common anti-virus products in practice. Consequently, we also compare it against the ten selected anti-virus scanners on our dataset. The detection performance of each scanner is again taken from the VirusTotal service. We run two experiments where we first consider all malware samples of our dataset and then only those samples provided by the Malgenome project [39]. We choose a false-positive rate of 1% for DREBIN which we think is sufficiently low for practical operation.

The results of the experiments are shown in Table II. The detection rate of the anti-virus scanners varies considerably. While the best scanners detect over 90% of the malware, some scanners discover less than 10% of the malicious samples, likely due to not being specialized in detecting Android malware. On the full dataset DREBIN provides the second best performance with a detection of 93.9% and outperforms 9 out of the 10 scanners. This observation is remarkable since, due to our test setting, at least two scanners should be able to detect each malware sample. Therefore, each sample has to be known for a certain amount time and most anti-virus scanners should be equipped with a corresponding signature. However, the automatically generated detection model of DREBIN proves to be more effective than the manually crafted signatures of many scanners. On the Malgenome dataset the anti-virus scanners achieve better detection rates, since these samples have been public for a longer period of time. Hence, almost all anti-virus scanners provide proper signatures for this dataset.

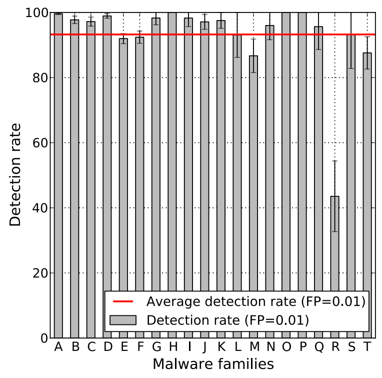
The false-positive rates of the anti-virus scanners range from 0% to 0.3% on our dataset of benign applications and thus are slightly below DREBIN's performance. Despite the vast number of available Android applications, the average user only installs some dozens of applications on his device. For example, according to Nielsen[1], a market research company, the average number of installed applications per smartphone in the U.S. has been 32 in 2011 and 41 in 2012. Consequently, we consider a false-positive rate of 1% still acceptable for operating DREBIN in practice.

3) Detection of malware families: Another important aspect that should be considered when testing the detection performance of a method is the balance of malware families in the dataset [32]. If the number of samples of certain malware

[1]Nielsen Report: "State of the Appnation – A Year of Change and Growth in U.S. Smartphones"



(a) Detection performance as ROC curve.



(b) Detection per malware family.

Id	**Family**	**#**	*Id*	**Family**	**#**
A	FakeInstaller	925	*K*	Adrd	91
B	DroidKungFu	667	*L*	DroidDream	81
C	Plankton	625	*M*	LinuxLotoor	70
D	Opfake	613	*N*	GoldDream	69
E	GingerMaster	339	*O*	MobileTx	69
F	BaseBridge	330	*P*	FakeRun	61
G	Iconosys	152	*Q*	SendPay	59
H	Kmin	147	*R*	Gappusin	58
I	FakeDoc	132	*S*	Imlog	43
J	Geinimi	92	*T*	SMSreg	41

(c) Top malware families in our dataset.

Fig. 4: Detection performance of DREBIN and related approaches.

families is much larger than of other families the detection result mainly depends on these families. To address this problem one can use the same number of samples for each family. However, this leads to a distribution that significantly differs from reality. Instead we evaluate the detection performance for each of the 20 largest malware families separately. The family names and the number of samples for each family can be found in Table 4(c) and the detection performance of DREBIN for each family is illustrated in Figure 4(b).

DREBIN is able to reliably detect all families with an average accuracy of 93% at a false-positive rate of 1%. In particular, all families show a detection rate of more than 90%, where three of them can be identified perfectly (H, O, P). There is only one malware family which cannot be reliably detected by DREBIN. This family is Gappusin (R) and we examine its low performance in the next section. It should be pointed out that there seems to be no dependency between the size of a malware family and its detection rate as long as the number of samples is sufficiently high and allows the SVM to generalize its features.

4) Detection of unknown malware families: DREBIN uses known malware for learning its detection model. It is thus important to assess how many samples of a family need to be known to reliably detect this family. To study this issue, we conduct two additional experiments where we limit the number of samples for a particular family in the training set. In the first experiment we provide no samples of the family, corresponding to a totally unknown malware strain. In the second experiment, we put 10 randomly chosen samples of the family back into the training set, thus simulating the starting spread of a new family.

The results of the two experiments are shown in Figure 5, where the detection rate is shown for 0 and 10 available samples in the training set for each family. If no samples are available for learning, it is difficult for DREBIN to detect a family, as no discriminative patterns can be discovered by the SVM. However, only very few samples are necessary to generalize the behavior of most malware families. With only 10 samples in the training set, the average detection performance increases by more than 25 percent. Three families can even be detected perfectly in this setting. The reason for this is that members of a certain families are often just repackaged applications with slight modifications. Due to the generalization which is done by the SVM it is therefore possible to detect variations of a family even though only a very small set of samples is known.

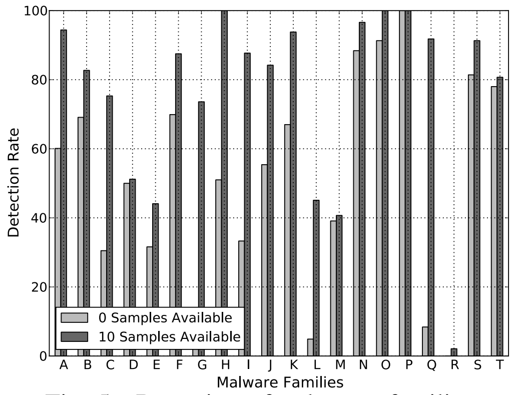


Fig. 5: Detection of unknown families.

In summary, DREBIN provides an effective detection of Android malware and outperforms related detection approaches as well as several anti-virus scanners. While DREBIN can not spot unknown malware from the very start, only few samples of each family are required for achieving a reliable detection.

C. Explanations

Apart from its detection performance a strength of DREBIN lies in its ability to explain the obtained results. This allows us to check whether the extracted features which contribute to the detection fit to common malware characteristics. In this section we first take a look at four popular malware families and analyze how features with high weights allow conclusions to be drawn about their behavior. We then inspect false positives and false negatives of DREBIN in detail.

1) Explanation for malware families: To study the explanations provided by DREBIN we consider four well-known malware families, namely FakeInstaller, GoldDream [23], GingerMaster [22] and DroidKungFu [24]. For each sample of these families we determine the features with the highest

Malware family	Top 5 features		
	Feature s	Feature set	Weight w_s
FakeInstaller	`sendSMS`	S_7 Suspicious API Call	1.12
	`SEND_SMS`	S_2 Requested permissions	0.84
	`android.hardware.telephony`	S_1 Hardware components	0.57
	`sendTextMessage`	S_5 Restricted API calls	0.52
	`READ_PHONE_STATE`	S_2 Requested permissions	0.50
DroidKungFu	`SIG_STR`	S_4 Filtered intents	2.02
	`system/bin/su`	S_7 Suspicious API calls	1.30
	`BATTERY_CHANGED_ACTION`	S_4 Filtered intents	1.26
	`READ_PHONE_STATE`	S_2 Requested permissions	0.54
	`getSubscriberId`	S_7 Suspicious API calls	0.49
GoldDream	`sendSMS`	S_7 Suspicious API calls	1.07
	`lebar.gicp.net`	S_8 Network addresses	0.93
	`DELETE_PACKAGES`	S_2 Requested permission	0.58
	`android.provider.Telephony.SMS_RECEIVED`	S_4 Filtered intents	0.56
	`getSubscriberId`	S_7 Suspicious API calls	0.53
GingerMaster	`USER_PRESENT`	S_4 Filtered intents	0.67
	`getSubscriberId`	S_7 Suspicious API calls	0.64
	`READ_PHONE_STATE`	S_2 Requested permissions	0.55
	`system/bin/su`	S_7 Suspicious API calls	0.44
	`HttpPost`	S_7 Suspicious API calls	0.38

TABLE III: Top features for the malware families FakeInstaller, DroidKungFu, GoldDream and GingerMaster.

contribution to the classification decision and average the results over all members of a family. The resulting top five features for each malware family are shown in Table III. For clarity we presents the exact features rather then the explaining sentences introduced in Section II-D.

- *FakeInstaller* is currently the most widespread malware. The members of this family hide their malicious code inside repackaged versions of popular applications. During the installation process the malware sends expensive SMS messages to premium services owned by the malware authors. Even on the first sight, three of the extracted features indicate that the malware uses SMS functionality, where `android.hardware.telephony` is implicitly added to the manifest file as soon as an application requests permissions to use SMS functionality.
- *DroidKungFu* tries to exploit several vulnerabilities in earlier Android versions to gain root access and steal sensitive data from the device. Its intention to gain root access is reflected by the feature `system/bin/su`. The invocation of `getSubscriberId()` indicates that the malware tries to access sensitive data. The two intents `BATTERY_CHANGED_ACTION` and `SIG_STR` are filtered by a broadcast receiver component which is part of many DroidKungFu samples. Both intents are used as a trigger to start a malicious service in the background as soon as a certain event occurs, for instance a low battery status.
- *GoldDream* is a Trojan which monitors an infected device, collects sensitive data and records information from received SMS messages. The feature `Telephony.SMS_RECEIVED` directly hints us to the reading of SMS messages. After the malware has collected sufficient data, it sends the data to an external server, whose hostname is second ranked in the feature list. Furthermore, the malware is able to install and delete packages as well as to send SMS messages, which is also reflected in the extracted features.
- *GingerMaster* is also a Trojan application which is often bundled with benign applications and tries to gain root access, steals sensitive data and sends it to a remote server. Similar to the DroidKungFu family the malware starts its malicious service as soon as it receives a `BOOT_COMPLETED` or `USER_PRESENT` intent. Again a significant part of this behavior can be reconstructed just by looking at the top features.

To study the contribution of the different feature sets to the detection of malware in general, we extract the top 5 features for all of the malware families in our dataset. The results are presented in Table IV. Although the requested permissions occur in the top features of all families, it is evident that this feature set alone is not sufficient to ensure a reliable detection. In particular, each feature set occurs at least once in the table which clearly indicates that all sets are necessary for the detection of Android malware.

2) False and missing detections: We finally examine benign applications which are wrongly classified as malware by DREBIN. Similar to malicious applications, most of these samples use SMS functionality and access sensitive data, which is reflected in high weights of the corresponding features. Moreover, these samples often show only very little benign behavior and thus trigger false alarms. Fortunately, the ability of DREBIN to output explainable results can help the user to decide whether a suspicious looking functionality is indeed malicious or needed for the intented purpose of the application. A similar situation occurs when DREBIN classifies samples of the Gappusin family [19] as benign. Although it is in many cases possible to extract features which match the description

Feature sets	Malware families																			
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
S_1 Hardware components	✓			✓				✓					✓							
S_2 Requested permissions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_3 App components			✓	✓	✓					✓		✓		✓			✓			
S_4 Filtered intents		✓				✓	✓						✓			✓				
S_5 Restricted API calls	✓				✓															
S_6 Used permissions																				✓
S_7 Suspicious API calls	✓	✓		✓		✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	
S_8 Network addresses			✓					✓	✓	✓	✓				✓	✓		✓	✓	✓

TABLE IV: Contribution of the feature sets to the detection of malware families.

of the Gappusin family—amongst others the hostname of the external server—there are too few malicious features to identify the samples as malware. Gappusin mainly acts as a downloader for further malicious applications and thus does not exhibit common malicious functionality, such as theft of sensitive data.

D. Run-time Performance

While the computing power of mobile devices is rapidly increasing, it is still limited compared to regular desktop computers. Consequently, a detection method that is supposed to run directly on these devices has to carry out its task very efficiently.

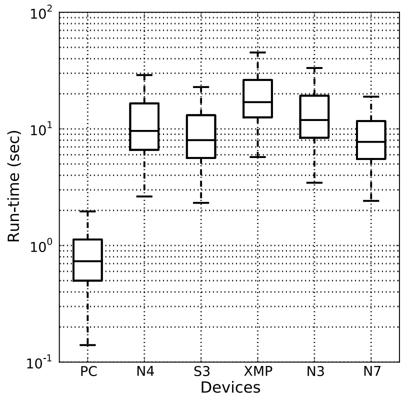


Fig. 6: Run-time performance of DREBIN.

To analyze the run-time of DREBIN we implement a standalone Android application that receives a learned detection model and is able to perform the detection process directly on the smartphone. The size of the downloaded model is only about 280 kbytes. Using this application we measure the run-time of DREBIN on different devices using 100 randomly selected popular applications from the Google Play Store. For this experiment, we choose devices which cover various widespread hardware configurations including four smartphone (Nexus 4, Galaxy S3, Xperia Mini Pro and Nexus 3), a tablet (Nexus 7) and a regular desktop computer (PC).

The results are presented in Figure 6. On average, DREBIN is able to analyze a given application in 10 seconds on the five smartphones. Even on older models, such as the Xperia Mini Pro, the method is able to analyze the application in less than 20 seconds on average. Overall, no analysis takes longer than 1 minute on all devices. On the desktop computer (2.26 GHz Core 2 Duo with 4GB RAM) DREBIN achieves a remarkable analysis performance of 750 ms per application, which enables scanning 100,000 applications in less than a day.

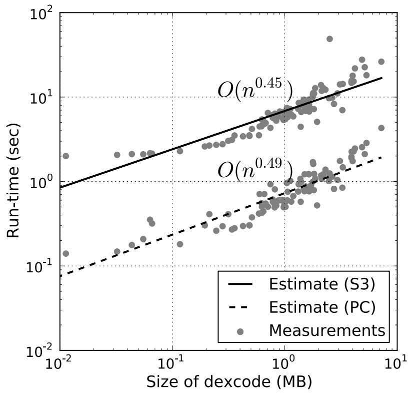


Fig. 7: Detailed run-time analysis of DREBIN.

A detailed run-time analysis for the desktop computer and the Galaxy S3 smartphone is presented in Figure 7, where the run-time per application is plotted against the size of the analyzed code. Surprisingly, on both devices DREBIN attains a sublinear run-time, that is, its performance increases with $O(\sqrt{m})$ in the number of analyzed bytes m. Apparently, the number of features does not increase linearly with the code and thus larger applications do not necessarily contain more features to analyze.

From this evaluation, we conclude that DREBIN does not only reliably detect malicious applications but is furthermore capable to perform this task in a time which clearly meets practical requirements.

IV. LIMITATIONS

The previous evaluation demonstrates the efficacy of our method in detecting recent malware on the Android platform. However, DREBIN cannot generally prohibit infections with malicious applications, as it builds on concepts of static analysis and lacks dynamic inspection. In particular, transformation attacks that are non-detectable by static analysis, as for example based on reflection and bytecode encryption [see 30], can hinder an accurate detection. To alleviate the absence of a dynamic analysis, DREBIN extracts API calls related to obfuscation and loading of code, such as `DexClassLoader.loadClass()` and

`Cipher.getInstance()`. These features enable us to at least spot the execution of hidden code—even if we cannot further analyze it. In combinations with other features, DREBIN is still able to identify malware despite the use of some obfuscation techniques.

To avoid crafting detection patterns manually, we make use machine learning for generating detection models. While learning techniques provide a powerful tool for automatically inferring models, they require a representative basis of data for training. That is, the quality of the detection model of DREBIN critically depends on the availability of representative malicious and benign applications. While it is straightforward to collect benign applications, gathering recent malware samples requires some technical effort. Fortunately, methods for offline analysis, such as DroidRanger [40], AppsPlayground [29] and RiskRanker [21], might help here to automatically acquire malware and provide the basis for updating and maintaining a representative dataset for DREBIN over time.

Another limitation which follows from the use of machine learning is the possibility of mimicry and poisoning attacks [e.g., 25, 27, 35]. While obfuscation strategies, such as repackaging, code reordering or junk code insertion do not affect DREBIN, renaming of activities and components between the learning and detection phase may impair discriminative features [30, 38]. Similarly, an attacker may succeed in lowering the detection score of DREBIN by incorporating benign features or fake invariants into malicious applications [25, 27]. Although such attacks against learning techniques cannot be ruled out in general, the thorough sanitization of learning data [see 7] and a frequent retraining on representative datasets can limit their impact.

V. RELATED WORK

The analysis and detection of Android malware has been a vivid area of research in the last years. Several concepts and techniques have been proposed to counter the growing amount and sophistication of this malware. An overview of the current malware landscape is provided in the studies of Felt et al. [16] and Zhou & Jiang [39].

1) Detection using static analysis: The first approaches for detecting Android malware have been inspired by concepts from static program analysis. Several methods have been proposed that statically inspect applications and disassemble their code [e.g., 12, 13, 15, 21]. For example, the method Kirin [13] checks the permission of applications for indications of malicious activity. Similarly, Stowaway [15] analyzes API calls to detect overprivileged applications and RiskRanker [21] statically identifies applications with different security risks. Common open-source tools for static analysis are Smali [17] and Androguard [10], which enable dissecting the content of applications with little effort.

Our method DREBIN is related to these approaches and employs similar features for identifying malicious applications, such as permissions, network addresses and API calls. However, it differs in two central aspects from previous work: First, we abstain from crafting detection patterns manually and instead apply machine learning to analyze information extracted from static analysis. Second, the analysis of DREBIN is optimized for effectivity *and* efficiency, which enables us to inspect application directly on the smartphone.

2) Detection using dynamic analysis: A second branch of research has studied the detection of Android malware at runtime. Most notably, are the analysis system TaintDroid [11] and DroidScope [37] that enable dynamically monitoring applications in a protected environment, where the first focuses on taint analysis and the later enables introspection at different layers of the platform. While both systems provide detailed information about the behavior of applications, they are technically too involved to be deployed on smartphones and detect malicious software directly.

As a consequence, dynamic analysis is mainly applied for offline detection of malware, such as scanning and analyzing large collections of Android applications. For example, the methods DroidRanger [40], AppsPlayground [29], and CopperDroid [31] have been successfully applied to study applications with malicious behavior in different Android markets. A similar detection system called Bouncer is currently operated by Google. Such dynamic analysis systems are suitable for filtering malicious applications from Android markets. Due to the openness of the Android platform, however, applications may also be installed from other sources, such as web pages and memory sticks, which requires detection mechanisms operating on the smartphone.

ParanoidAndroid [28] is one of the few detection systems that employs dynamic analysis and can spot malicious activity on the smartphone. To this end, a virtual clone of the smartphone is run in parallel on a dedicated server and synchronized with the activities of the device. This setting allows for monitoring the behavior of applications on the clone without disrupting the functionality of the real device. The duplication of functionality, however, is involved and with millions of smartphones in practice operating ParanoidAndroid at large scale is technically not feasible.

3) Detection using machine learning: The difficulty of manually crafting and updating detection patterns for Android malware has motivated the application of machine learning. Several methods have been proposed that analyze applications automatically using learning methods [e.g., 2, 26, 33]. As an example, the method of Peng et al. [26] applies probabilistic learning methods to the permissions of applications for detecting malware. Similarly, the methods Crowdroid [4], DroidMat [36], Adagio [20], MAST [5], and DroidAPIMiner [1] analyze features statically extracted from Android applications using machine learning techniques. Closest to our work is DroidAPIMiner [1] which provides a similar detection performance to DREBIN on recent malware. However, DroidAPIMiner builds on a k-nearest neighbor classifier that induces a significant runtime overhead and impedes operating the method on a smartphone. Moreover, DroidAPIMiner is not designed to provide explanations for its detections and therefore is opaque to the practitioner.

Overall, previous work using machine learning mainly focuses on an accurate detection of malware. Additional aspects, such as the efficiency and the explainability of the detection, are not considered. We address these aspects and propose a method that provides an effective, efficient and explainable detection of malicious applications.

VI. Conclusion

Android malware is a new yet fast growing threat. Classic defenses, such as anti-virus scanners, increasingly fail to cope with the amount and diversity of malware in application markets. While recent approaches, such as DroidRanger [40] and AppPlayground [29], support filtering such applications off these markets, they induce a run-time overhead that is prohibitive for directly protecting smartphones. As a remedy, we introduce DREBIN, a lightweight method for detection of Android malware. DREBIN combines concepts from static analysis and machine learning, which enables it to better keep pace with malware development. Our evaluation demonstrates the potential of this approach, where DREBIN outperforms related approaches and identifies malicious applications with few false alarms.

In practice, DREBIN provides two advantages for the security of the Android platform: First, it enables efficiently scanning large amounts of applications, such as from third-party markets. With an average run-time of 750 ms per application on a regular computer, it requires less than a day to analyze 100,000 unknown applications. Second, DREBIN can be applied directly on smartphones, where the analysis can be triggered when new applications are downloaded to the device. Thereby, DREBIN can protect users that install applications from untrusted sources, such as websites and third-party markets.

Although DREBIN effectively identifies malicious software in our evaluation, it exhibits the inherent limitations of static analysis. While it is able to detect indications of obfuscation or dynamic execution, the retrieved code is not accessible by the method. A similar setting has been successfully tackled for the analysis of JavaScript code [see 9] and dynamically triggering the static analysis of DREBIN whenever new code is loaded seems like a promising direction of future work.

Dataset

To foster research in the area of malware detection and to enable a comparison of different approaches, we make the malicious Android applications used in our work as well as all extracted feature sets available to other researchers under *http://user.cs.uni-goettingen.de/~darp/drebin*.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments and Yan Chen for shepherding the paper. The authors acknowledge funding from BMBF under the project PROSEC (FKZ 01BY1145).

References

[1] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in android," in *Proc. of International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2013.

[2] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 73–84.

[3] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communication of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[4] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proc. of ACM Worksgop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011, pp. 15–26.

[5] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Mast: Triage for market-scale mobile malware analysis," in *Proc. of ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2013.

[6] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. MIT Press, 1989.

[7] G. Cretu, A. Stavrou, M. Locasto, S. Stolfo, and A. Keromytis, "Casting out demons: Sanitizing training data for anomaly sensors," in *Proc. of IEEE Symposium on Security and Privacy*, 2008, pp. 81–95.

[8] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines*. Cambridge, UK: Cambridge University Press, 2000.

[9] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser JavaScript malware detection," in *Proc. of USENIX Security Symposium*, 2011.

[10] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," in *Presentation at Black Hat Abu Dhabi*, 2011.

[11] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 393–407.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proc. of USENIX Security Symposium*, 2011.

[13] W. Enck, M. Ongtang, and P. D. McDaniel, "On lightweight mobile phone application certification," in *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2009, pp. 235–245.

[14] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *Journal of Machine Learning Research (JMLR)*, vol. 9, pp. 1871–1874, 2008.

[15] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2011, pp. 627–638.

[16] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proc. of ACM Worksgop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011, pp. 3–14.

[17] J. Freke, "An assembler/disassembler for android's dex format," Google Code, http://code.google.com/p/smali/, visited February, 2013.

[18] "Mobile threat report 2012 q3," F-Secure Response Labs, 2012.

[19] 2012, http://www.naked-security.com/malware/Android.Gappusin/.

[20] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call

graphs," in *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*, Nov. 2013, pp. 45–54.

[21] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proc. of International Conference on Mobile Systems, Applications, and Services (MOBISYS)*, 2012, pp. 281–294.

[22] X. Jiang, "Security alert: Gingermaster," 2011, http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/.

[23] ——, "Security alert: Golddream," 2011, http://www.csc.ncsu.edu/faculty/jiang/GoldDream/.

[24] ——, "Security alert: New droidkungfu variant," 2011, http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/.

[25] J. Newsome, B. Karp, and D. Song, "Paragraph: Thwarting signature learning by training maliciously," in *Recent Adances in Intrusion Detection (RAID)*, 2006, pp. 81–105.

[26] H. Peng, C. S. Gates, B. P. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 241–252.

[27] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *Proc. of IEEE Symposium on Security and Privacy*, 2006, pp. 17–31.

[28] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2010.

[29] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *Proc. ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[30] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: evaluating android anti-malware against transformation attacks," in *Proc. of ACM Symposium on Information, Computer and Communications Security(ASIACCS)*, 2013.

[31] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," in *Proc. of European Workshop on System Security (EUROSEC)*, April 2013.

[32] C. Rossow, C. Dietrich, C. Gier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen, "Prudent practices for designing malware experiments: Status quo and outlook," in *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[33] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspective combining risks and benefits," in *Proc. of ACM symposium on Access Control Models and Technologies (SACMAT)*, 2012, pp. 13–22.

[34] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proc. of IEEE Symposium on Security and Privacy*, 2010, pp. 305–316.

[35] S. Venkataraman, A. Blum, and D. Song, "Limits of learning-based signature generation with adversaries," in *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2008.

[36] 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 *Proc. of Asia Joint Conference on Information Security (Asia JCIS)*, 2012, pp. 62–69.

[37] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis," in *Proc. of USENIX Security Symposium*, 2012.

[38] M. Zheng, P. P. Lee, and J. C. Lui, "ADAM: an automatic and extensible platform to stress test android anti-virus system," in *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012, pp. 82–101.

[39] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. of IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

[40] 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 *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2012.