Hugin: A Scalable Hybrid Android Malware Detection System

Dominik Teubert, Johannes Krude, Samuel Schüppen, Ulrike Meyer
Department of Computer Science
RWTH Aachen University
Aachen, Germany
Email: {teubert,krude,schueppen,meyer}@itsec.rwth.aachen.de

***Abstract*—Mobile operating systems are a prime target of today's malware authors and cyber criminals. In particular, Google's Android suffers from an ever increasing number of malware attacks in the form of malicious apps. These typically originate from poorly policed third-party app stores that fail to vet the apps prior to publication. In this paper, we present *Hugin*, a machine learning-based app vetting system that uses features derived from dynamic, as well as static analysis and thus falls into the scarcely studied class of hybrid approaches. *Hugin* is unique with respect to using IPC/RPC monitoring as source for dynamically extracted features. Furthermore, *Hugin* uses a short (and yet effective) feature vector that leads to a high efficiency in training as well as classification. Our evaluation shows that *Hugin* achieves a detection accuracy of up to 99.74% on an up-to-date data set consisting of more than 14,000 malware samples and thus, is easily capable of competing with other current systems.**

Keywords–mobile malware detection; app vetting; machine-learning.

I. INTRODUCTION

Smartphones are omnipresent in our society. According to a recent study, 72% of the adults in the U.S. and 60% in Europe own a smartphone [1]. Google's Android is particularly popular with a leading market share of 86.2% [2] at the time of writing. Similar to its competitors, Android does not only provide an operating system, but a complete eco-system for app development and distribution. Unlike platforms, such as Apples's iOS, Android does not restrict users to the official app store. This lead to the emergence of a number of third-party app stores, gaining popularity especially in world regions such as Russia and Asia. These alternative markets are not as tightly regulated as the official Google Play store and are therefore often used for malware distribution. It is estimated that up to 3-7% of the available apps in Asian app stores are malware, compared to only 0.1% malicious apps in Google's official Play store [3]. Google recently warned that the chance of installing a potentially malicious app is ten times larger outside the official store [4]. In Russia, up to 8.3% of the apps installed from outside Google's Play store are potentially harmful [5].

The operators of the official app stores fight malware with different approaches. Google introduced the Bouncer [4] [6], a semi-automated approach that utilizes mainly dynamic analysis for malware detection. Apple even performs a manual review of the apps submitted to their app store. Although third-party app stores have an equally strong interest in keeping their market places free of malware, the numbers above show that many of them are poorly policed. Large enterprises that have a mobile device management (MDM) solution in place create an additional barrier to keep their devices safe. Mobile devices under MDM are often restricted to company operated app stores that have a particularly high security standard but a limited amount of apps to choose from. However, there is no established procedure how to vet apps before they are published in such a store for the first time. This demonstrates the importance of scalable automated mobile malware detection for third-party app store operators and large companies alike.

This gap is filled by malware detection systems that are based on machine learning and therefore are able to detect yet unknown threats. Existing approaches use various types of features derived from static analysis (e. g., [7]–[10]), dynamic analysis (e. g., [11] [12]), or even both (e. g., [13]). However, while inter process communication has previously been used to analyze the malicious behavior of a specific app [14]–[16], none of the prior malware detection system uses higher level Inter Process Communication (IPC)/Remote Procedure Calls (RPC) (monitoring as source for dynamically extracted features. In this paper, we introduce *Hugin*, a novel malware detection system based on a hybrid of static and dynamic features. Hugin is unique with respect to using IPC/RPC as feature source and has a very good detection capability comparable to the best already existing mobile malware detection systems. In particular, *Hugin* has the following properties:

Hybrid Detection: *Hugin* uses as feature vector containing features derived by static as well as features derived by dynamic analysis. We evaluate the static and the dynamic part of the feature vector separately and show that *Hugin* benefits from the hybrid approach in terms of detection accuracy.

IPC-based Features: Although IPC is heavily used on Android, to the best of our knowledge, *Hugin* is the first approach using higher level IPC/RPC calls as a source for dynamic features in the context of an Android malware detection system.

Reliable Features: Android malware detection that relies on static features derived from disassembled or decompiled code often suffers from degraded detection performance due to obfuscation. In contrast, *Hugin* relies mainly on static features derived from parts of the APK that are hard to obfuscate.

Compact Feature Vector: *Hugin* makes use of a comparatively low number of (static and dynamic) features selected by feature engineering. This short and thus compact feature vector allows for efficient training ($< 32\ s$) and classification ($< 3\ ms$).

Strong Detection Performance: *Hugin* shows an excellent detection rate of about 99% (with a false-positive rate well below 1%) on the well established Drebin data set (covering the time period from 2008–2012) and even better accuracy on the more recent, newly generated *Hugin* data set.

The remainder of this paper is structured as follows: The most closely related work is summarized in Section II. A sys-

 ISBN: 978-1-61208-582-1

tem overview on *Hugin* with details on the feature extraction and the training and classification is given in Section III. We present the results of our evaluation of *Hugin* in Section IV. We conclude in Section V.

II. Related Work

In the following section, we summarize the most relevant mobile malware detection approaches from related work. We focus on those approaches that are similar to *Hugin* in the sense that they use either static or dynamic analysis to extract features and machine learning techniques for detection or classification. Note that a systematic comparison of detection results between the proposed systems is only possible for systems that made their data sets publicly available (such as Drebin [8]) or base their evaluation on such data sets (such as Droidsieve [10] and DroidScribe [12]).

1) Detection based on static analysis: Many approaches from the field of machine learning aided mobile malware detection use features that are derived from static analysis. For unobfuscated malware, static features are typically easy and computationally inexpensive to extract from APK files and therefore allow for fast and scalable solutions. Additionally, many static features are well established and understood (e. g., Android permissions). One of the earlier approaches of static mobile malware detection was proposed by Peng et al. [7]. The authors used probabilistic generative models such as Naive Bayes to rank Android apps according to their asscociated risk for the user. For training these models, Peng et al. relied mainly on the requested permissions. With DroidSIFT [9] Zhang et al. proposed a system that extracts API dependency graphs to reconstruct Android app semantics. Graph similarity metrics are then used to obtain a classification decision and thus to distinguish benign from malicious apps. Following this procedure DroidSIFT achieves a detection rate of 93% on a malware set of 2200 samples. Drebin [8] provides a static detection method that extracts features such as permissions, filtered intents, API calls, and URLs. For classification Drebin also uses SVMs, but constructs the vector space in such a way that the system can present explanations for the detection decision to the user. Furthermore, its lightweight nature allows for detection on the end-user device. The authors of Drebin provide a public data set consisting of 5560 malicious apps, which is also used to evaluate *Hugin*. On this data set, Drebin achieves a detection rate of up to 94% based on 545,000 features. One of the most recent approaches that was proposed in this area is DroidSieve [10]. DroidSieve aims for classifying obfuscated as well as unobfuscated Android malware solely with the help of static features. Obfuscation-invariant features as well as artifacts indicating obfuscation are used to enable the system to also classify obfuscated samples correctly. The elaborated feature engineering results in a promising accuracy of up to 99.64% on the Drebin data set using 22,584 features. Using feature selection as an additional step, DroidSieve reduces the number of features to 859 with a slight drop in accuracy (99.57%).

2) Detection based on dynamic analysis: The landscape of related approaches that derive features from dynamic analysis is smaller. This is mainly due to high demands of dynamic analysis regarding the setup of the emulation environment and the hardware requirements when performing analysis at scale. Note that there are also various systems such as Droidscope [14], AppsPlayground [15], and Copperdroid [16] that assist dynamic analysis of mobile malware, but do not aim for automated detection. Among the first systems that used dynamic features is Crowdroid [11]. Burgueara et al. used system call invocations counts as features and subsequently performed clustering using the k-means algorithm. The correct label for each cluster (benign or malicious) is determined using a crowdsourcing-based approach, assuming that a large enough user base will reveal the significantly smaller malicious cluster. Most recently Dash et al. proposed DroidScribe [12], a system that focuses on classifying Android malware samples into families. DroidScribe exclusively uses run-time behavior such as system call traces, file/network access, and Binder communication to construct dynamic features. Using different flavors of SVMs the system achieves a classification accuracy of up to 94%.

3) Detection based on hybrid analysis: Hybrid mobile malware detection, i. e., the combination of static and dynamic features for later classification, is even less comprehensively studied. The only closely related approach to *Hugin* is Marvin [13]. Lindorfer et al. extract a pile of 450,000 static and dynamic features and use SVMs as well as linear classifier to calculate a malice score for each app. Their best configuration achieves a convincing detection rate of 98.24% at a low false-positive rate. Marvin also uses feature selection as additional step in its training procedure, ending up with 27,808 highest ranked features (and a strong emphasis on the dynamic features). In contrast to the fetch-all feature extraction of Marvin, *Hugin* uses feature engineering and ends up with far less features (about 2,000) at a comparable accuracy. Furthermore, while Marvin relies on traditional dynamic features such as file/network operations and data leakage, *Hugin* incorporates IPC/RPC-based features for the first time in the field of Android malware detection.

III. System Overview

Since *Hugin* is a machine learning-based approach, it operates in two phases: the training phase and the classification phase. The training phase takes labeled data sets of benign and malicious apps as input and results in a trained model. The classification phase represents the actual operational mode: apps that are submitted to an app store are processed and a binary decision on basis of the pre-trained model is made. Depending on the outcome further actions can be necessary, e.g. the rejection of the app. To detect Android malware, *Hugin* analyzes each app to get a comprehensive representation of its behavior. While many proposed approaches focus either on static or dynamic analysis *Hugin* combines both techniques to soften the limitations of either of these lines of work. Figure 1 shows an overview of *Hugin*'s detection approach and its different stages. The key aspects are:

Static feature extraction. Our approach uses static analysis to inspect each Android installation package. *Hugin* focuses on features that can be extracted reliably, even for many obfuscated malware samples.

Dynamic feature extraction. The dynamic analysis part of *Hugin* relies on monitoring the inter-process communication (IPC) of each sample at runtime. IPC is heavily used on Android and almost all potentially harmful functionality of apps has to pass this interface. Thus, a detailed profile of the

 ISBN: 978-1-61208-582-1

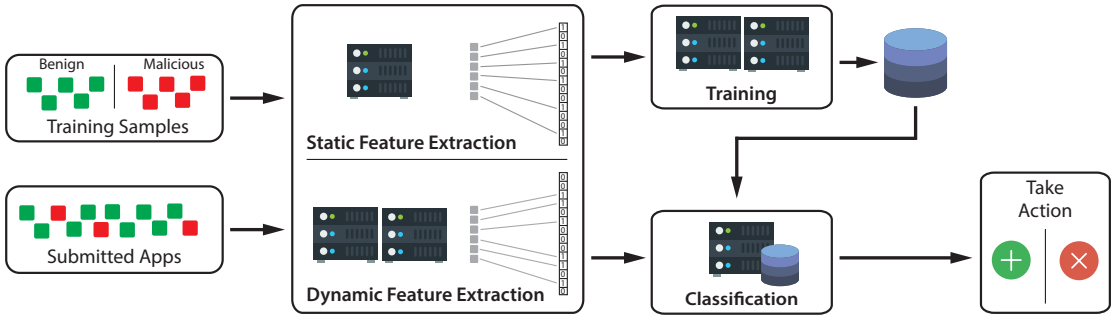


Figure 1. *Hugin* system overview.

runtime behavior of the analyzed apps is created and is used to derive features from it.

Training & Classification. For training and classification SVMs with different kernels are used. SVMs showed outstanding classification performance in a variety of application areas. The utilization of kernels provides a high degree of flexibility. Due to the comparably short length of our feature vector even computationally expensive kernels such as the Radial Basis Function (RBF) kernel can efficiently be used for classification.

A. Static Feature Extraction

Static analysis is frequently used for malware detection purposes and eases automation and scalability through its lightweight nature. However, static analysis also has several downsides. Properties that can be statically extracted can be differentiated into those which are particularly prone to obfuscation techniques and those which are reliably extractable in most cases. The higher the semantic level of information that is gained through static analysis, the higher the chance that this analysis can be hindered by malware. In particular, complex code recovery through disassembling or decompilation is often affected by obfuscation mechanisms. Detection approaches that rely solely on features derived from static analysis are therefore easy to circumvent by sophisticated malware. In contrast, *Hugin* confines itself to those static properties that can reliably be extracted from Android install packages (APK files) and uses these to derive features. On Android, the `Manifest` file is a particularly good source for static analysis, since it contains essential information about each app such as permissions, activities, services, broadcast receivers, and intent-filters. Furthermore, the `Manifest` is mandatory for the installation of new apps and has a pure declarative character, both usually preventing it from being obfuscated. *Hugin* therefore primarily relies on a comparatively small set of 1326 static features that originate from the `Manifest` file. Since the size of the feature vector is a crucial factor for training and classification efficiency, our approach supports efficient classification even with complex algorithms such as RBF-SVMs.

Specifically, we extract the following static features:

Permissions. Android makes use of permissions to separate apps according to their privileges. Permissions are therefore app-specific and are actively granted once by the user at installation time. Access to many sensitive system resources such as the location- or telephony-subsystem is controlled via permissions. Malware depends on using such resources to suceed and therefore usually requests many security-related permissions [17] [18].

Hardware components. In the case apps want to use hardware components such as the camera, the microphone, or the GPS this has to be declared in the `Manifest`. Many types of malware, in particular spyware, heavily depend on using such hardware features. The request of multiple sensitive hardware components can therefore be indicative for malicious behavior.

Intent-filters. Intents on Android allow apps to listen for different events that are propagated through the system. Malware often uses intents to trigger certain malicious actions, e. g., starting a background service after the `BOOT_COMPLETED` intent is received.

Activity count. The primary goal of malware is to execute its malicious payload. Most malware is therefore kept rather simple regarding its interface to the user. We represent this common property with the activity count, being 1 if the app has > 3 activities and 0 otherwise.

Service count. The service count follows the same logic. Malware frequently makes use of background services to perform malicious actions without the user's awareness. However, malware tends to put its malicious code into a single service, a high number of services is rather indicative for a complex benign app. We therefore add a 1 to the binary feature vector if the app has > 2 services and 0 otherwise.

Third-party libraries. The only features that are not directly derived from the `Manifest` are the utilized third-party libraries. For extraction, the `androguard` framework is used. Our assumption is that some third-party libraries can be particularly indicative for adware.

Automating the static analysis of APK files to extract the 1326 features does not pose a major challenge. There exist well established tools included in the Android SDK and from the open-source community that were used within *Hugin* (in particular `aapt` and `androguard`). For each analyzed app, a binary feature vector indicating the presence or absence of each feature is created. This static feature vector is later merged with the dynamic feature vector yielding a comprehensive vector for training and classification.

B. Dynamic Feature Extraction

Static analysis often does not suffice to detect sophisticated malware that uses code obfuscation or dynamic code loading. To reveal such hidden malicious behavior malware analysts often make use of dynamic analysis. Monitoring the runtime behavior of apps can provide additional insights that

TABLE I. EXAMPLES FOR EACH SET OF FEATURES.

Static features
Permissions (518 features)
`android.permission.INTERNET`
`android.permission.READ_SMS`
`android.permission.REBOOT`
Hardware components (104 features)
`android.hardware.MICROPHONE`
`android.hardware.LOCATION`
`android.hardware.CAMERA`
Intent-filters (638 features)
`android.intent.action.SERVICE_STATE`
`android.intent.action.BOOT_COMPLETED`
`android.intent.action.SCREEN_OFF`
Third-party libraries (62 features)
`com.google.android.maps`
`android.software.device_admin`
`sonymobile.enterprise.api_1`
Dynamic features
System services (195 features)
`android.os.IServiceManager`
`android.app.IAlarmManager`
`android.os.IPowerManager`
Remote Procedure Calls (516 features)
`sendText`
`getSubscriberId`
`startService`
Dynamic permissions (58 features)
`android.permission.RECEIVE_SMS`
`android.permission.WAKE_LOCK`
`android.permission.SEND_SMS`

might disclose potentially harmful actions, but does also raise a number of new challenges. In contrast to static analysis, dynamic analysis has very high demands regarding the analysis environment in terms of hardware, performance, and setup. As described below, *Hugin* meets these challenges and complements the static features with dynamic features derived from monitored IPC/RPC logs.

1) IPC on Android: The architecture of the Android operating system heavily relies on IPC/RPC mechanisms to provide a variety of functionality to apps. Namely, Android utilizes Binder for IPC, a reimplementation of a protocol dating back to OpenBinder [19]. Various important features of modern smartphones such as sending SMS, accessing the GPS location, and taking pictures are made accessible to developers via Binder and its RPC interface [20] [21]. Most aspects of the implementation details of Binder IPC are hidden from the Android developers using this RPC interface and corresponding Java APIs that built on it. However, whenever the funtionality of a (sensitive) Android system service is being used, the Binder interface has to be passed. Note that it is irrelevant if some functionality provided by a system service is used in the context of a Java-written app or using native code. The Binder is involved in both cases. Therefore, monitoring the IPC interface allows to create detailed profiles of the behavioral aspects of apps at run-time. For this reason, IPC monitoring was already used, e.g., in [14], [16] with the goal of supporting the analysis of the malicious behavior of a specific app. However, to the best of our knowledge, we are the first who use IPC/RPC calls on Android to derive features for an automated malware detection system.

2) Dynamic Analysis Environment: One challenge in dynamic malware analysis in general is that malware tries to detect that it runs in a sandbox. The same holds for malicious apps: malicious apps try to detect if they are running in an emulator and change their behavior if they do. Thus there is a complete line of research on how malware can detect that it is running in an emulator and how to make a malcious app believe that it runs on an actual device (e.g. [22] [23]). While this line of work is orthogonal to our work, we tried to incorporate some of these findings into the Android Virtual Devices (AVDs) used during our dynamic analysis. In particular, we used the list of properties that can be queried via the Android API to perform sandbox detection published in [22] to modify the AVDs. We also tried to mimic the actual usage of the emulated device by installing some common apps (e.g. signed-in Facebook and Twitter apps) and storing data such as some contacts in the phone book.

Besides considering sandbox detection, the stimulation of dynamically tested apps to increase code coverage is a much discussed topic. In recent years sophisticated stimulation approaches were proposed [24] [25]. *Hugin* incorporates some simpler heuristics to trigger typical malware behavior. In particular, we reboot the emulator to trigger the commonly used `BOOT_COMPLETED` intent-filter, send and receive SMS, perform a phone call, and modify time and date settings. Additionally, we make use of the `monkey`, an application exerciser included in the Android SDK that allows injecting random events for a specific duration. Each `monkey` phase runs for 180 sec in our test setup, the complete dynamic analysis of each app takes about 10 mins.

3) IPC Monitoring: To monitor IPC on Android we implemented *BTrace* (short for Binder Trace). *BTrace* is a modified Android Emulator to capture Android Binder inter-process communication events using virtual machine introspection. These captured events range from low-level Binder `ioctl`'s to high level remote procedure calls, intent broadcasts, content provider access and the dynamic evaluation of used permissions. *BTrace* produces both human-readable and machine-readable output, the former intended for manual inspection, the latter for automatic analysis. Although we used DroidScope [14] (that publicly provides only very basic hooking mechanisms) as a starting point, *BTrace* does not reuse anything from DroidScope with the exception of emulator system call hooks and emulator memory access routines. Unlike DroidScope, *BTrace* derives all monitored binder events from a kernel system call view. To evaluate binder remote procedure calls and higher level events, system call arguments and return values are used. The structures on how to interpret this data were extracted, in large parts automatically, from the Android source code. *BTrace* employs an automata describing the Android process creating and naming behavior. This automata is used to determine the point of time at which the name of an app may securely be read from user-space. Events are attributed to app-name by remembering the once read app-name for a process and the transitive hull of its child processes.

 ISBN: 978-1-61208-582-1

For different kinds of actions, *BTrace* dynamically analyses whether permissions are needed to perform these actions. Unfortunately, the Android permission specification does not exist in the form of a formal specification but only through an implementation spread across the Android Java and C++ source code. To evaluate dynamic permission usage, *BTrace* employs a permission specification obtained through executing PScout [26], a tool that performs static program analysis on the Android source code to generate the corresponding specification.

Specifically, we extract the following dynamic features:

Used system services. For each analyzed app we monitor which system services are used via the Binder interface during run-time. The combination of several sensitive system services can already be indicative for suspicious behavior.

Remote Procedure Calls. The specific method calls that are performed on each system service give even deeper insights into the run-time behavior. Since many system services provide a broad set of methods the actually used methods allow a better differentiation between benign and potentially harmful actions.

Dynamically used permissions. Using the permission specification obtained from PScout we are able to monitor the permissions that are actually used at run-time. The rationale behind this is that many benign apps statically request too many permissions that are not or only very rarely used. In contrast, aggressive malware will very likely use many of the statically requested permissions even in the limited timeframe of analysis.

C. Training & Classification

As described before, the app store vetting scenario *Hugin* aims for has exceptionally high demands regarding the detection capabilities of deployed systems. In particular, detection engines that guard an app store from unwanted software should be able to detect previously unknown threats. These requirements naturally suggest the application of machine learning and binary classification in particular. To this end, we utilized Support Vectors Machines (SVMs) [27] [28] for all evaluated classification tasks. Specifically, we implemented the classification part of *Hugin* using the efficient LIBSVM library [29].

SVMs are non-probabilistic binary classifiers which were successfully applied in a variety of application areas (e.g. in computational biology and chemistry [30]). Besides their strong classification performance [31], SVMs provide further interesting properties: flexibility through utilization of kernels [28], strong theoretic guarantees regarding the generalization performance [32], and the support of one-class classification through an extension [33]. Kernels are particularly interesting because they allow efficient non-linear classification. The "kernel trick" performs an implicit mapping from the input vector space into a (higher or even infinite-dimensional) feature vector space. Data points that are not linearly separable in the input vector space may be separable in this higher-dimensional vector space. *Hugin* was evaluated with the standard linear kernel and the Gaussian Radial Basis Function (RBF) kernel. In case of the RBF kernel the linear inner product $K(x, x') = x \cdot x'$ is replaced with $K(x, x') = \exp(\frac{-|x-x'|^2}{\gamma})$ within the dual representation of the SVM. Note that the RBF kernel is computationally much more expensive than the linear kernel. Here, *Hugin* profits from its comparatively short feature vector of over all 2095 binary features allowing efficient classification even for complex kernels.

TABLE II. TOP 10 FAMILIES OF THE EVALUATED DATA SETS.

Drebin data set			*Hugin* data set		
Id	**Family**	**# samples**	**Id**	**Family**	**# samples**
A	FakeInstaller	925	A	FakeInstaller	5724
B	DroidKungFu	667	D	Opfake	1078
C	Plankton	625	K	SmsSpy	735
D	Opfake	613	L	Dowgin	708
E	GingerMaster	339	M	RuSMS	438
F	BaseBridge	330	N	SmsStealer	274
G	Iconosys	152	O	FakeToken	261
H	Kmin	147	P	Lotoor	233
I	FakeDoc	132	F	BaseBridge	185
J	Geinimi	92	Q	Boxer	123

IV. Evaluation

In this section, we present the results of our evaluation of the performance of *Hugin*. In particular, we detail the evaluation methodology and data sets used, present the detection performance of *Hugin* for the static part, the dynamic part and the hybrid feature vector, compare the performance to prior approaches as far as possible, and detail the training and classification efficiency of *Hugin*. Note that a systematic and sound comparison between systems with respect to their detection capabilities is only possible if the systems are evaluated on the same data sets. This is only possible for systems that published the data sets on which they evaluated themselves (such as Drebin [8]) or systems that in turn based their evaluation on such public data sets (such as Droidsieve [10] and DroidScribe [12]). We therefore compare the detection performance of *Hugin* on the Drebin data set to the performance of these systems on the same data set only.

A. Data Sets and Methodology

1) Data sets: We evaluate *Hugin* on two different malicious data sets, the Drebin dataset [8] containing 5560 malicious samples and a newly assembled dataset of 14,043 malicious samples referred to as *Hugin* dataset throughout the rest of this paper. To compare our approach to prior work, we used the Drebin data set [8], which covers the time period between August 2010 and October 2012. Note that we were able to extract features from 5317 samples only. The remaining 243 were either corrupted APK files and failed already in the static analysis or failed in the dynamic analysis because the could not be installed on the emulated device. Additionally, the more recent *Hugin* data set covers the time-period between January 2015 and September 2016. To compose this *Hugin* data set we used the VirusTotal intelligence search, querying the mentioned time period and requesting at least 35 AV matches to ensure the sample is indeed malware. Table II shows the top 10 families of the Drebin and the *Hugin* data set. While some families such as FakeInstaller and Opfake are still popular in the newer *Hugin* data set, others such as DroidKungFu and Plankton dropped out of this top-list. Last but not least, we composed a benign *Hugin*-b data set. To this end, we downloaded 14,068 popular apps from the official Google Play store, assuming that the fraction of potentially harmful apps in

TABLE III. SVM EVALUATION FOR THE DREBIN DATA SET.

Linear Kernel $C = 1$				RBF Kernel $\gamma = 0.03125, \nu = 0.03125$			
Features	**TPR**	**FPR**	**ACC**	**Features**	**TPR**	**FPR**	**ACC**
hybrid	97.21%	1.03%	98.49%	hybrid	97.57%	0.52%	98.95%
static	93.19%	1.39%	97.12%	static	95.81%	1.21%	97.97%
dynamic	93.56%	5.69%	94.11%	dynamic	88.60%	3.21%	94.54%

TABLE IV. SVM EVALUATION FOR *HUGIN* DATA SET.

Linear Kernel $C = 1$				RBF Kernel $\gamma = 0.03125, \nu = 0.0039062$			
Features	**TPR**	**FPR**	**ACC**	**Features**	**TPR**	**FPR**	**ACC**
hybrid	99.70%	0.54%	99.58%	hybrid	99.66%	0.19%	99.74%
static	99.14%	1.02%	99.06%	static	99.57%	0.48%	99.55%
dynamic	98.92%	5.14%	96.89%	dynamic	95.67%	3.32%	96.18%

TABLE V. DATASETS USED IN THE EVALUATION OF HUGIN.

Data set name	**Ground truth**	**# samples**
Drebin	malware	5,317
Hugin	malware	14,043
Hugin-b	benign	14,068

TABLE VI. COMPARISON OF RELATED APPROACHES EVALUATING THE DREBIN DATA SET (STATING THE BEST MENTIONED CONFIGURATION).

Approach	**Features**	**Best ACC**	**# features**
Drebin [8]	static	∼96.50%	∼545,000
DroidSieve [12]	static	99.64%	∼22,500
Hugin-static	static	97.97%	1326
Droidscribe [10]	dynamic	94.00%	254
Hugin-dynamic	dyanmic	94.54%	769
Hugin	hybrid	98.95%	2,095

the most popular apps is particularly low. Note that the *Hugin*-b data set was used as the benign training set in all performed experiments due to the fact that the Drebin-b data set is not publicly available. Table V summarizes the sizes of all data sets used for evaluation.

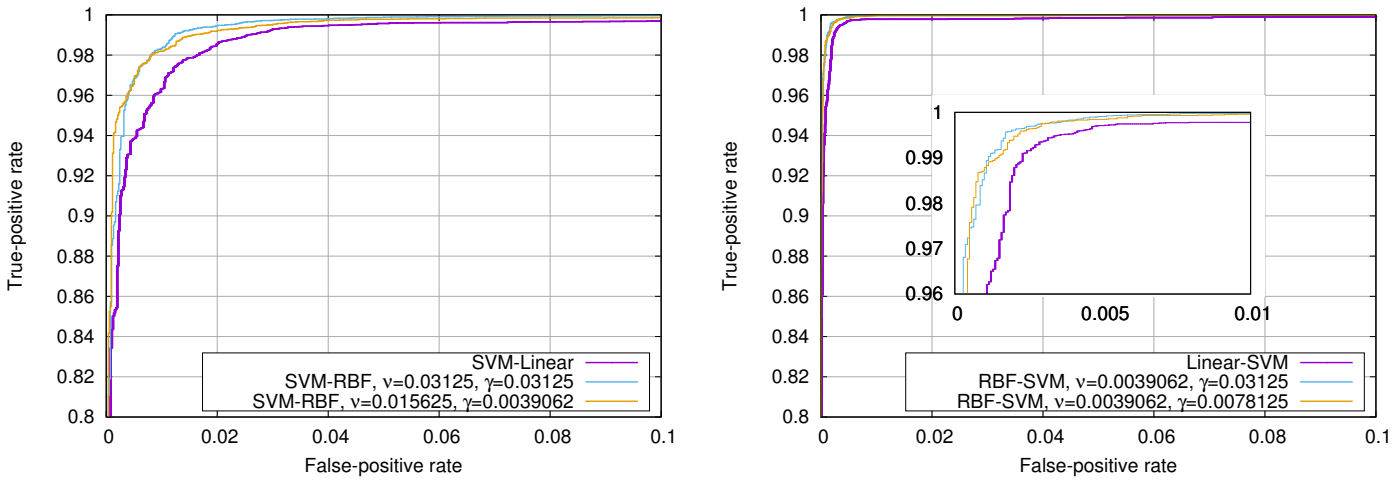
2) Methodology: The detection performance of *Hugin* was measured by performing various experiments. In these experiments all relevant performance measures were calculated by splitting each data set into a training partition (66% of the samples) and a validation partition (33% of the samples). To this end, we applied repeated random subsampling and averaged our results over 10 runs. We used standard performance measures like the true-positive rate (TPR), the false-positive rate (FPR), and the accuracy (ACC) to assess the performance of *Hugin* and to be able to compare our approach to others. Additionally, we used Receiver Operating Characterstic (ROC) curves to visualize different parameter combinations [34]. In our first series of experiments, we evaluated *Hugin* against two malicious data sets, the publicly available Drebin data set and the newly assembled *Hugin* data set. For classification we tested SVMs with linear kernel and SVMs with RBF kernel. In case of the RBF kernel we performed a grid search to determine the kernel parameter γ and ν. We also evaluated the static feature vector, the dynamic feature vector, and the hybrid feature vector (concatenation of static and dynamic vector) separately.

B. Overall Detection Performance

Table III shows the results for the Drebin data set for the best kernel parameter determined through grid search. Overall, *Hugin* achieves an accuracy of just below 99% on the Drebin data set, with the RBF kernel showing superior TPR and FPR compared to the linear kernel. Interestingly, in case of the RBF kernel considering only the static feature vector yields far better results than considering only the dynamic feature vector, while the results are more balanced for the linear kernel. However, in both cases the hybrid feature vector performs best, underpinning the assumed benefits of hybrid mobile malware detection. In case of the public Drebin data set we are able to directly compare *Hugin* to related approaches. While there are minor methodical differences between the detection approaches (e. g., regarding the fraction of the samples that are used for training and validation), the overall trend should be unaffected. Section IV-A2 shows *Hugin*'s excellent accuracy compared to the most closely related approaches that have evaluated the Drebin data set. Only the purely static DroidSieve [10] approach that is optimized for obfuscated malware achieves an even higher accuracy, but requires 16 times more features to achieve its best performance (see Section IV-A2). Note that *Hugin* is on par with the highly feature intensive Drebin and the Droidscribe approach when considering only the static or dynamic feature vector, respectively. The overall superiority of *Hugin* can therefore be attributed to the combination of both analysis techniques.

Table IV summarizes the detection results on the more up-to-date *Hugin* data set. Overall, the detection performance is even better than on the Drebin data set. Both the Linear-SVM and the RBF-SVM achieve an accuracy of over 99.5%, with the RBF-SVM performing best regarding the FPR. Evaluated

 ISBN: 978-1-61208-582-1



(a) ROC for Drebin data set.

(b) ROC for *Hugin* data set.

Figure 2. ROC curves for both data sets.

individually, we again observe that the dynamic feature vector performs worse than the static feature vector and that this effect is more pronounced for the RBF-SVM. However, it also becomes evident on the *Hugin* data set that the detection performance profits from the hybrid detection approach.

C. Detection Performance of Malware Families

The considerable better detection performance on the *Hugin* data set compared to the Drebin data set is also illustrated with the ROC curves in Figure 2. For both data sets the curves for the Linear-SVM and for the RBF-SVM parameter optimizing the ACC and TPR, respectively, are plotted. In addition to the better detection performance on the *Hugin* data set, the main finding is the consistently better performance of the RBF-SVM compared to the Linear-SVM on both data sets. We also assume that the considerably better performance on the *Hugin* data set can mainly be attributed to the benign data set used for training: Since it covers a similar time span as the malicious *Hugin* data set, it is easier for the classifier to distinguish these apps than on the considerably older Drebin data set (note that the Drebin-b data set is not publicly available). A lesson learned therefore is, that the benign and malicious training data set should always stem from a similar time period.

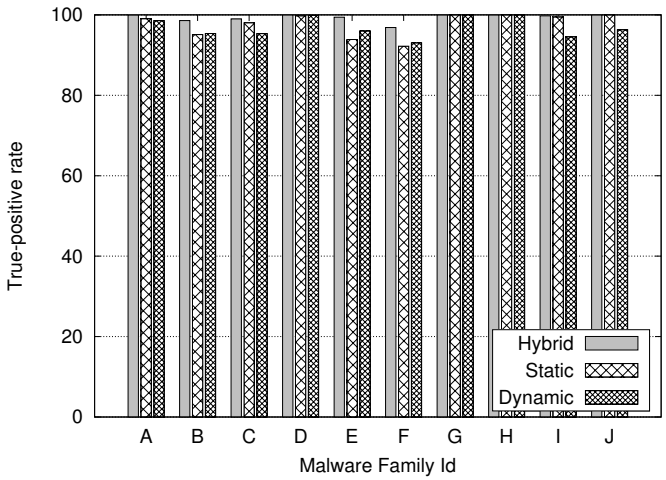
Figure 3 shows the detection rates of *Hugin* for the top 10 families of the Drebin and the *Hugin* data set. Compared to the Drebin approach, *Hugin* performs similar or better on the 10 most frequent malware families with an average detection rate of 99.35%. For the families E and F (GingerMaster and BaseBridge) that were particularly hard to detect for Drebin (detection rates of below 93%) our approach achieves significantly better detection rates of 99.41% and 96.86%, respectively. The authors of Drebin also reported a particular bad detection rate for the Gappusin family (not ranked top 10) and explained this result with the low number of extractable features. Interestingly, we can replicate this result when considering solely the static feature vector (51.44% TPR) or the dynamic feature vector (68.58% TPR). However, the hybrid feature vector achieves a compelling detection rate of 93.95%. This result once again indicates that the combination of static and dynamic features can help detecting mobile malware that is otherwise hard to detect. The top 10 families of the *Hugin* data set show even higher detection rates with an average of 99.51%. When considering only the dynamic feature vector we can observe a significantly higher average detection rate of 99.09% on the *Hugin* data set (compared to 96.89% on the Drebin data set). This phenomenon can easily be explained with age of the data sets: Since the Drebin data set is much older, the dynamic analysis can extract less features because, e. g., command and control server are put offline and therefore less behavior is shown during the analysis. Consistently, the gap in the average detection rate is smaller when using only the static feature vector for classification (99.10% on the *Hugin* data set, 97.74% on the Drebin data set). For the families that are included in both data sets (A, D, and F) the BaseBridge family (Id F) is particularly interesting. With a detection rate of below 93% in the original Drebin paper, BaseBridge is among the families that are most difficult to detect. *Hugin* achieves a detection rate of 96.86% for the BaseBridge samples in the Drebin data set and even 97.51% detection rate for the samples included in the *Hugin* data set, while the detection rate for the dynamic feature vector again increased notably on the newer data set. This leads us to conclude that the hybrid *Hugin* approach shows its strongest performance for most recent malware samples that emit a considerable amount of dynamic behavior the system can profit from.

D. Efficiency

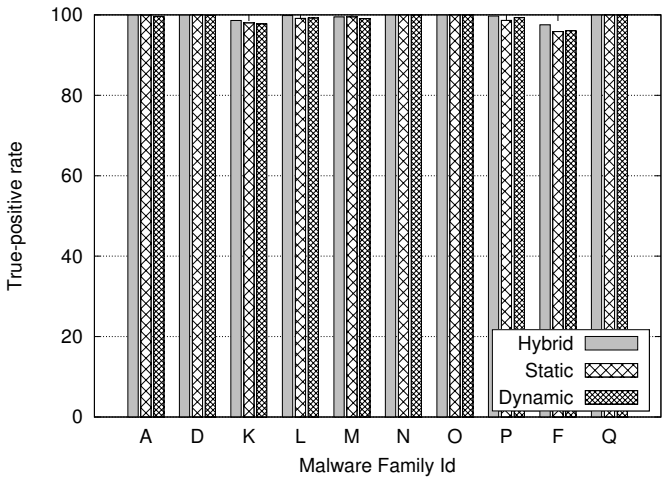
The feature extraction part of *Hugin* consists of a dynamic as well as a static module. While the dynamic analysis of each app is quiet costly (8-10 minutes, see Section III), the actual extraction of the feature vector from the log data is negligible (52.37 ms per app, averaged over the Drebin data set). As expected the static feature extraction shows far better performance, with an average of only 55.39 ms for the entire analysis of each app.

The advantage of the short feature vector of *Hugin* (which is one of its strength) shows best in the training and classifi-

 ISBN: 978-1-61208-582-1



(a) Drebin data set.



(b) Hugin data set.

Figure 3. Detection rates per malware family for the Linear-SVM.

cation performance. To get the most meaningful numbers, we measured the performance for the experiment with the highest number of feature vectors (training with the *Hugin* data set, i.e. about 28,000 feature vectors) and averaged over 10 runs. Using the RBF kernel the training of the SVM took 31.56 sec, while the classification of the 9559 apps in the validation set took 22.05 sec (i. e., 2.31 ms on average per app). The linear kernel shows even better performance. In this case the training took 24.78 sec and the classification 17.57 sec, i. e., only 1.84 ms per app.

Note that all numbers were measured on quiet dated desktop hardware (Intel i7-2600@3.40GHz) and therefore leave much room for improvements (either through more powerful hardware or through persistent use of parallelization).

V. Conclusion and Future Work

In this paper, we present *Hugin*, a hybrid and scalable Android malware detection system. We show how lightweight static analysis and complex dynamic analysis can be combined to create a comprehensive yet compact feature vector. *Hugin* achieves an accuracy of up to 99.74% on an up-to-date data set with far less features than related approaches. Our evaluation proves that the system profits significantly from the hybrid approach, both in terms of overall detection performance and in terms of detection performance for malware families that are particularly hard to detect. In particular, our dynamic feature extraction that relies on monitored interprocess communication proved to be a meaningful addition. Each of the individual components of *Hugin* is subject to continuous advances of the academic community, which could also improve *Hugin*. Static analysis could benefit from more elaborated feature engineering that allow better detection of obfuscated malware samples. Dynamic analysis could be enhanced by incorporating more complex stimulation techniques that increase code coverage. Furthermore, the post-processing of the results which can include report generation for analysts was not addressed so far and is another direction of future work.

Acknowledgments

We want to thank VirusTotal as well as the authors of Drebin for providing us with data sets to evaluate our approach. Additionally, we want to thank Madebyoliver from www.flaticons.com for his marvelous icon sets.

References

[1] J. Poushter, “Smartphone ownership and internet usage continues to climb in emerging economies,” Online, 2016, URL: http://www.pewglobal.org/2016/02/22/smartphone-ownership-and-internet-usage-continues-to-climb-in-emerging-economies/ [retrieved: July, 2017].

[2] Gartner Inc., “Gartner says five of top 10 worldwide mobile phone vendors increased sales in second quarter of 2016,” Online, Aug. 2016, URL: http://www.gartner.com/newsroom/id/3415117 [retrieved: July, 2017].

[3] F-Secure, “Mobile threat report,” Online, 2014, URL: https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf [retrieved: July, 2017].

[4] Google Inc., “Android security 2015 year in review,” Tech. Rep., 2016.

[5] ——, “Android security 2014 year in review,” Tech. Rep., 2015.

[6] H. Lockheimer, “Android and security,” online, 2012, URL: http://googlemobile.blogspot.de/2012/02/android-and-security.html [retrieved: July, 2017].

[7] H. Peng et al., “Using probabilistic generative models for ranking risks of android apps,” in Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012, pp. 241–252.

[8] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, “DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.” in NDSS, 2014.

[9] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, “Semantics-aware android malware classification using weighted contextual api dependency graphs,” in Proceedings of ACM CCS, 2014, pp. 1105–1116.

[10] G. Suarez-Tangil et al., “Droidsieve: Fast and accurate classification of obfuscated android malware,” in CODASPY. ACM, 2017, pp. 309–320.

[11] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, “Crowdroid: behavior-based malware detection system for android,” in Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2011, pp. 15–26.

[12] S. K. Dash et al., “Droidscribe: Classifying android malware based on runtime behavior,” in IEEE Symposium on Security and Privacy Workshops. IEEE Computer Society, 2016, pp. 252–261.

 ISBN: 978-1-61208-582-1

[13] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis." in COMPSAC. IEEE Computer Society, 2015, pp. 422–433.

[14] L. K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). USENIX, 2012, pp. 569–584.

[15] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in Proceedings of the third ACM conference on Data and application security and privacy. ACM, 2013, pp. 209–220.

[16] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors." in NDSS, 2015.

[17] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in Proceedings of IEEE S&P. IEEE Computer Society, 2012.

[18] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspective combining risks and benefits," in Proceedings of the 17th ACM symposium on Access Control Models and Technologies. ACM, 2012, pp. 13–22.

[19] D. K. Hackborn, "Openbinder," online, 2005.

[20] K. Yaghmour, Embedded Android: Porting, Extending, and Customizing. O'Reilly Media, Inc., 2013.

[21] N. Elenkov, Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press, 2014.

[22] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection." in ASIA CCS. ACM, 2014, pp. 447–458.

[23] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in Proceedings of the Seventh European Workshop on System Security. ACM, 2014, p. 5.

[24] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro, and S. Zanero, "Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications." CoRR, vol. abs/1402.4826, 2014.

[25] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda, "CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes," in Proceedings of the 20th International Conference on Financial Cryptography and Data Security (FC), 2016.

[26] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: analyzing the Android permission specification," in ACM Conference on Computer and Communications Security. ACM, 2012, pp. 217–228.

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

[28] C. Bishop, Bishop Pattern Recognition and Machine Learning. Springer, New York, 2001, p. 325ff.

[29] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," ACM Transactions on Intelligent Systems and Technology, vol. 2, 2011, pp. 27:1–27:27, software available at URL: http://www.csie.ntu.edu.tw/~cjlin/libsvm [retrieved: July, 2017].

[30] O. Ivanciuc, "Applications of support vector machines in chemistry," Reviews in computational chemistry, vol. 23, 2007, p. 291.

[31] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems," J. Mach. Learn. Res, vol. 15, no. 1, 2014, pp. 3133–3181.

[32] V. Vapnik, The nature of statistical learning theory. Springer Science & Business Media, 2013.

[33] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, and J. C. Platt, "Support vector method for novelty detection." NIPS, vol. 12, 1999, pp. 582–588.

[34] D. M. W. Powers, "Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation," Journal of Machine Learning Technologies, vol. 2, no. 1, 2011.

 ISBN: 978-1-61208-582-1