More Semantics More Robust: Improving Android Malware Classifiers

Wei Chen
University of Edinburgh, UK
wchen2@inf.ed.ac.uk

David Aspinall
University of Edinburgh, UK
david.aspinall@ed.ac.uk

Andrew D. Gordon
Microsoft Research
Cambridge, UK
University of Edinburgh, UK
andy.gordon@ed.ac.uk

Charles Sutton
University of Edinburgh, UK
csutton@inf.ed.ac.uk

Igor Muttik
Intel Security, UK
igor.muttik@intel.com

ABSTRACT

Automatic malware classifiers often perform badly on the detection of new malware, i.e., their robustness is poor. We study the machine-learning-based mobile malware classifiers and reveal one reason: the input features used by these classifiers can't capture general behavioural patterns of malware instances. We extract the best-performing syntax-based features like permissions and API calls, and some semantics-based features like happen-befores and unwanted behaviours, and train classifiers using popular supervised and semi-supervised learning methods. By comparing their classification performance on industrial datasets collected across several years, we demonstrate that using semantics-based features can dramatically improve robustness of malware classifiers.

Keywords

Mobile security; Android system; malware detection; machine learning

1. INTRODUCTION

The machine-learning-based classification plays an important role in automatic mobile malware detection. The main drawback is its poor robustness—the classification performance on the detection of new malware is bad [4]. Researchers have shown that well-trained classifiers can achieve good classification performance, e.g., precision as high as 99% and false positive ratio as low as 1% [3, 7, 42]. However, in these and most other studies, the training and testing data were collected in the same period and from the same source. These classifiers only presented good fits to training data. When these classifiers are applied in practice to detect new malware, the classification accuracy drops dramatically. A method adopted in industry to mitigate this problem is to replace some old training data by new data and re-train classifiers to maintain good classification performance. But it is hard to decide how much old data should be removed and what kind of new data should be added.

In this paper we ask whether it is possible to improve robustness of classifiers over time, by using more general and abstract features, rather than simply substituting new data for old training data. We want to figure out the main factor which affects robustness of mobile malware classifiers and develop an approach to improve it. The main contributions of this paper are as follows.

- We show that the known best-performing classifiers, e.g., those using API calls as input features, perform badly on the detection of new malware; in particular, the precision and recall respectively drop from around 95% and 99% on the validation dataset to on average 55% and 26% on the testing dataset.
- We compare the classification performance of classifiers which were trained using popular supervised and semi-supervised learning methods, and conclude that the L1-Regularized Linear Regression is the most robust method, i.e., showing better and balanced performance on the validation and testing datasets.
- We demonstrate that semantics-based features improve robustness dramatically, in particular, increasing the precision and recall on the testing dataset respectively to as high as 73% and 67%, which are respectively 18 and 41 points better than those using syntax-based features.

We train and test using Android apps from several industrial datasets. They were collected and investigated between 2011 and 2014 by third-party researchers and malware analysts from anti-virus vendors.

- **Training and Validation.** We collected 3,000 malware instances, which were released and identified between 2011 and 2013, and 3,000 benign apps published in the same period. They include all malware instances from Malware Genome Project [45] and most from Mobile-Sandbox [34]. These malware instances have been manually investigated and organised into around 200 families by third-party researchers and malware analysts [1, 2, 27]. They were divided into a training

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WiSec'16 , July 18-22, 2016, Darmstadt, Germany

© 2016 ACM. ISBN 978-1-4503-4270-4/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2939918.2939931

dataset and a validation dataset. Each of them consists of 1,500 malware instances across all families and 1,500 benign apps.

- **Testing.** We test using a collection of 1,500 malware instances, which were released and identified in 2014, and 1,500 benign apps published in the same year. These malware instances were from Intel Security and have been investigated by malware analysts. The collection of benign appls is disjoint from those used for training and validation. They were randomly chosen from benign apps supplied by Intel Security.

We want to experiment on small datasets before testing on market-scale datasets in further work. We found that when the training dataset contained more than 1,000 apps, a well-trained classifier performed stably; so, datasets containing thousands of apps are enough for our purpose. Since the distribution of malware in real world is unknown, for each dataset we simply put in the same number of samples and kept malware and benign half-and-half.

We report performance of classifiers which were trained using the following machine learning methods:

- *trees:* decision trees [31], random forest [12], and the adaptive boosting [22] using decision trees as the base estimators;
- *linear:* the L1-regularized linear regression [37] and support vector machines [35];
- *semi-supervised*: the work by Zhou et al. [44];
- *others: k*-nearest neighbours [5] and naive Bayes,

and the following features:

- *syntax-based:* permissions, actions, API calls, and keywords;
- *semantics-based:* reachables, happen-befores, and unwanted behaviours.

These features were directly extracted from the bytecode of Android apps using static analysis. All semantics-based features are based on an abstract model called behaviour automata, which are collections of finite control-sequences of actions, events, and annotated API calls, to approximate the behaviours of Android apps. We adopt the approach proposed in [17] to construct behaviour automata and learn unwanted behaviours from them. More details on the feature extraction are given in Section 2.

A classifier is considered *robust* if its classification performance is good and balanced on the validation and testing datasets. Formally, we measure robustness of classifiers by calculating the F_β-measure [32] of F_1-scores of precision and recall on these two datasets. We trained 56 classifiers by using the above methods and features. The comparison between these classifiers demonstrates that semantics-based features improve robustness of malware classifiers. The details are given in Figure 3 and Table 3.

Malicious behaviours in a group of apps might be innocuous in another, e.g., sending text messages is normal for messaging apps but suspicious for E-reader apps; so, to further improve robustness we train and test cluster-specific classifiers. That is, apps are organised into small groups by using clustering methods; then, one classifier is trained for each group. The evaluation shows that robustness of these cluster-specific classifiers is better than general classifiers, especially, when semantics-based features are applied in the clustering process.

These evaluations confirm our intuition: semantics-based features capture general behavioural invariants in malware, which leads to better classification performance on the detection of new malware than that of classifiers using syntax-based features. We believe that by using more fine-grained semantics-based features better classification performance can be achieved.

Related Work.

Machine learning methods have been applied in Android malware detection for some time. Researchers have tested various supervised learning methods and different kinds of features. For example, the tool DroidAPIMiner [3] uses API calls as input features and relies on the KNN algorithm; the method Drebin [7] trains an SVM classifier using a range of syntax-based features; Yerima et al. applied naive Bayes [41] and ensemble learning [40] in training; Gascon et al. [23] proposed to use graph kernel of embedded call graphs; Zhang et al. [43] exploited the edit distance between API dependency graphs; behaviour graphs were used in DroidMiner [39]; Yuan et al. [42] designed a good deep-learning-based classifier; Narudin et al. [28] compared several methods and concluded that random forest and naive Bayes have the best classification performance. Clustering methods were applied as well. For instance, the tool Dendroid [36] uses the cosine similarity between call graphs to group malware instances into families; similar ideas were applied in DroidLegacy [18] to detect piggybacking. Among others, various probabilistic models were developed to rank risks in apps. For example, Peng et al. [30] built models on naive Bayes; the tool MAST [13] exploits the multiple correspondence analysis to figure out the most indicative features.

All of these tools and methods were trying to obtain good fits to training data by combining different methods and features. Robustness of malware classifiers, in particular, the classifier specifically designed to detect new malware, has received much less consideration. An early investigation of effects on classifiers caused by new malware has been done by Allix et al. [4]. They concluded that training on a random set of known malware could lead to significantly biased results. This discovery is also confirmed in our study, i.e., robustness of classifiers using syntax-based features is poor. Our research is beyond this primitive investigation and demonstrates a promising method to improve robustness of classifiers.

2. FEATURES

Syntax-based features are the most popular and the best-performing features known for malware classifiers, including: meta-information of an app, e.g. permissions, actions, intents, etc., and specific strings in code, e.g., API calls, commands, keywords appearing in UI elements, URLs, fragments of bytecode, etc. We call features "semantics-based" when they start to relate syntax-based features using dependency relations, e.g., an API method is invoked before another, the data-flow from a variable to another, a call-back is triggered by an event, call-graphs, etc. In this sec-

tion, we will discuss and compare several syntax-based and semantics-based features.

2.1 Syntax-Based Features

An Android app consists of the manifest file *AndroidManifest.xml*, the bytecode *classes.dex*, the developer's signatures, libraries, and resources including: layouts, pictures, strings, etc. The manifest file specifies permissions requested by the app and components defined in the app. A component is often associated with actions which are requests or events it can deal with. By using the platform tool `aapt` we extract permissions and actions from the manifest file, and all strings defined in resources, from which we will choose keywords. We decompile the bytecode into assembly code by using the platform tool `dexdump`, from which we extract API calls.

2.1.1 Permissions

Permissions reflect resource requirements from an app. Although the developer can define their own permissions, we only care about system permissions which are pre-defined in the Android framework. We extract system permissions from the manifest file, e.g., INTERNET, ACCESS_FINE_LOCATION, CAMERA, etc. Around 200 system permissions govern more than 32,000 API methods [9]. To invoke a permission-governed API method, the developer has to specify its corresponding permission in the manifest file; otherwise, the app will crash at runtime. However, an app might request more permissions than it actually needs, so-called over-privileged [19, 21]. Thus, the list of system permissions requested by an app is a lightweight but very coarse characterisation of its behaviour.

2.1.2 Actions

Actions denote what kind of requests or events an app can deal with. For example, the following fragment of the manifest file tells us: a receiver component is defined in this app; it can deal with the action SMS_RECEIVED.

```
<receiver android:name="com.example.Receiver" >
  <intent-filter>
    <action android:name=
      "android.provider.Telephony.SMS_RECEIVED"/>
  </intent-filter>
</receiver>
```

We are interested in actions because a lot of identified malware instances will exploit specific actions. For example, an instance in the malware family Zitmo [27] will intercept an incoming SMS message to obtain the user's online transaction number, i.e., the unwanted behaviour is triggered by the action SMS_RECEIVED; the unwanted behaviours in the malware families Arspam [34] and Ginmaster [45] are triggered by the action BOOT_COMPLETED, i.e., the device finishes the booting; some instances in the malware families Anserverbot and Basebridge [45] load classes from hidden payloads when a USB mass storage is connected, i.e., the action UMS_CONNECTED.

Developers are allowed to define their own actions to implement communications between components within the same app. Since these developer-defined actions are too specific to be used as training features, we only extract from the manifest file system actions which are pre-defined in the Android framework. Around 800 system actions were collected from more than 10,000 sample apps.

2.1.3 API Calls

API calls appearing in code tell us what an app can possibly do. We collected more than 52,000 API calls by going through the assembly code of more than 10,000 sample apps. For example, from the following assembly code,

```
#1 : (in Lcom/example/main/Main;)
 name : 'getPhoneNumber'
 type : '()Ljava/lang/String;'


|0000: invoke-virtual {v3},
  Lcom/example/main/Main;.getBaseContext
|0003: move-result-object v1
|0004: const-string v2, "phone"
|0006: invoke-virtual {v1, v2},
  Landroid/content/Context;.getSystemService
|0009: move-result-object v0
|000a: check-cast v0,
  Landroid/telephony/TelephonyManager;
|000c: invoke-virtual {v0},
  Landroid/telephony/TelephonyManager;.getLine1Number
|000f: move-result-object v1
|0010: return-object v1
```

we extract the API calls `Context.getSystemService` and `TelephonyManager.getLine1Number` by looking for the instructions `invoke-*`.

The list of API calls is the best-performing feature known for malware classifiers. By carefully selecting salient API calls, combining with other syntax-based features, and choosing suitable machine learning methods, the precision of classifiers can usually reach as high as 99% and the false positive ratio is maintained as low as 1% [3, 7]. However, API calls have two drawbacks.

- It contains "noise" caused by the dead code and libraries, in particular, advertisement libraries [3].

- It can't characterise more sophisticated app behaviours. This is needed in practice: some malicious behaviours only arise when some API methods are called in certain orders [16, 25, 39].

These drawbacks will result in overfitting to training data. Accordingly, the performance on the detection of new malware is poor, as we will show later.

2.1.4 Keywords

We extract nouns from strings which are defined in resources of Android apps, so-called keywords. These keywords will be presented to the user in UI elements at runtime. They reflect what an app declares to do. For instance, the keywords "photo", "gallery", and "camera" often appear in a Photo Editor app and the keywords "weather", "city", and "temperature" are often seen in a Weather Forecast app.

These keywords are more precise than those extracted from the descriptions of apps. For each app on the official Android market—Google Play, a description explaining its functionality is supplied by its developer. Researchers have studied how to organise Android apps into groups by using the keywords extracted from these descriptions and how to identify the outliers in each group, e.g., abnormal usages of APIs [24]. However, most malware instances were collected from alternative Android markets. Their descriptions might not exist or are not written in English. They often contain a lot of redundant words, which are added to boost the appearance in search results.

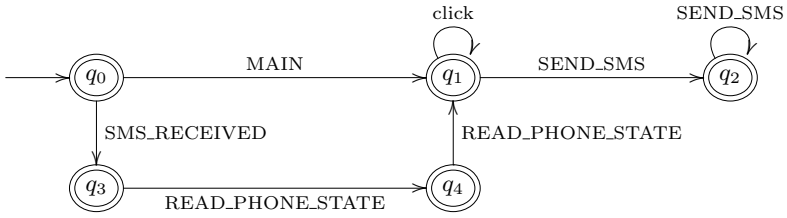

Figure 1: An example behaviour automaton.

Human-authored description	Learned unwanted behaviours in regular expressions
Arspam. Sends spam SMS messages to contacts on the compromised device.	1. BOOT_COMPLETED . SEND_SMS
Anserverbot. Downloads, installs, and executes payloads.	1. UMS_CONNECTED . LOAD_CLASS* . (ACCESS_NETWORK_STATE | READ_PHONE_STATE | INTERNET) . (ACCESS_NETWORK_STATE | READ_PHONE_STATE | INTERNET | LOAD_CLASS)*
Basebridge. Forwards confidential details (SMS, IMSI, IMEI) to a remote server.Downloads and installs payloads.	1. UMS_CONNECTED . (INTERNET | LOAD_CLASS | READ_PHONE_STATE | ACCESS_NETWORK_STATE)+
Cosha. Monitors and sends certain information to a remote location.	1. MAIN . click . (click | ACCESS_FINE_LOCATION | DIAL)* . DIAL . (click | ACCESS_FINE_LOCATION | DIAL)* . (INTERNET | ϵ) 2. SMS_RECEIVED . (INTERNET | ACCESS_FINE_LOCATION)+
Droiddream. Gains root access, gathers information (device ID, IMEI, IMSI) from an infected mobile phone and connects to several URLs in order to upload this data.	1. PHONE_STATE . (ACCESS_NETWORK_STATE | READ_PHONE_STATE+ . INTERNET) . (ACCESS_NETWORK_STATE | INTERNET)*
Geinimi. Monitors and sends certain information to a remote location.Introduces botnet capabilities with clear indications that command and control (C&C) functionality could be a part of the Geinimi code base.	1. ϵ | MAIN . click+ . VIBRATE . (click | VIBRATE)* . RESTART_PACKAGES . (MAIN . (click | VIBRATE)* . RESTART_PACKAGES)* 2. BOOT_COMPLETED . (ACCESS_NETWORK_STATE | click | INTERNET | RESTART_PACKAGES | ACCESS_FINE_LOCATION)+
Ggtracker. Monitors received SMS messages and intercepts SMS messages.	1. MAIN . READ_PHONE_STATE 2. SMS_RECEIVED . SEND_SMS
Ginmaster. Sends received SMS messages to a remote server. Downloads and installs applications without user concern.	1. BOOT_COMPLETED . LOAD_CLASS 2. MAIN . SEND_SMS
Spitmo. Filters SMS messages to steal banking confirmation codes.	1. NEW_OUTGOING_CALL . READ_PHONE_STATE . INTERNET . (INTERNET | ϵ)
Zitmo. Opens a backdoor that allows a remote attacker to steal information from SMS messages received on the compromised device.	1. SMS_RECEIVED . SEND_SMS 2. MAIN . READ_PHONE_STATE 3. MAIN . SEND_SMS

Table 1: Human-authored descriptions versus learned unwanted behaviours.

2.2 Semantics-Based Features

We approximate an app's behaviour by an automaton, i.e., a collection of finite control-sequences of events, actions, and annotated API calls. Some API calls might indicate the same behaviour, for instance, `getDeviceId`, `getLine1 Number`, and `getSimSerialNumber` are all related to the behaviour of reading phone state; so we aggregate API calls into permission-like phrases and abstract automata by substituting phrases for API calls, so-called *behaviour automata* [17].

An example behaviour automaton is given in Figure 1. It tells us: this app has two entries which are respectively specified by the actions MAIN and SMS_RECEIVED; it will collect information like the phone state, then send SMS messages out; it can deal with the interaction from the user, e.g., clicking a button, touching the screen, long-pressing a picture, etc., which is denoted by the word "click". All states in this automaton are accepting states since any prefix of an app's behaviours is one of its behaviours as well.

We have designed and implemented a static analysis tool to construct behaviour automata. This tool models complex real-world features of the Android framework, including: inter-procedural calls, callbacks, component life-cycles, inter-component communications, multiple threads, multiple entries, nested classes, and runtime-registered listeners. We don't model registers, fields, assignments, operators, pointer-aliases, arrays or exceptions. The choice of which aspects to model is a trade-off between efficiency and precision. In our implementation, we use an extension of permission-governed API methods generated by PScout [9] as the annotations. The Android platform tool `dexdump` is used to decompile the bytecode into assembly code, from which we construct automata.

From behaviour automata we produce the following features: reachables, happen-befores, and unwanted behaviours.

2.2.1 Reachables

Reachables denote the labels on edges which can be reached along a path from one of the entries in a behaviour automaton. For instance, all labels on the edges of the automaton in Figure 1 are reachables. They are more precise than permissions, actions, and API calls appearing in code. This semantics-based feature removes the "noise" caused by dead code and libraries. It reflects what an app can actually do but no order.

2.2.2 Happen-Befores

The happen-before denotes that something happens before another in a behaviour automaton. For example, the following pairs:

```
(MAIN, click), (SMS_RECEIVED, SEND_SMS),
(MAIN, SEND_SMS), (SMS_RECEIVED, click),
(SMS_RECEIVED, READ_PHONE_STATE),
(READ_PHONE_STATE, SEND_SMS),
(READ_PHONE_STATE, click), (click, SEND_SMS),
```

are happen-before features extracted from the automaton in Figure 1. These pairs characterise some interesting malicious behaviours which the reachables can't capture. For instance, the pair `(SMS_RECEIVED, SEND_SMS)` is a characterisation of a common malicious behaviour shared by malware instances in the family Zitmo [27]: obtaining the online transaction number from the incoming messages then sending it out by SMS messages to a specific phone number, to finish the online transaction instead of the real user.

In general, one can extract n-tuples as features from behaviour automata, i.e., things happening in certain orders. But, this will introduce a lot of redundant sequences, e.g., a "click" list, which waste the space for other more indicative features. Also, we found that constructing triples was already too expensive.

The happen-befores are less precise than pairs of sources and sinks produced by the data-flow analysis tools like FlowDroid [8] or Amandroid [38]. However, compared with generating data-flow models, it is much easier to produce happen-befores for a large number of apps.

2.2.3 Unwanted Behaviours

An unwanted behaviour is a common sub-automaton which is shared by malware instances but rarely identified in benign apps. As an example, let us consider a malware family called Ggtracker [2]. A brief human-authored description of this family produced by Symantec is as follows.

> *It sends SMS messages to a premium-rate number. It monitors received SMS messages and intercepts SMS messages. It may also steal information from the device.*

One unwanted behaviour we have constructed from malware instances in this family can be expressed as the regular expression: SMS_RECEIVED.SEND_SMS. It denotes the behaviour of sending an SMS message out *immediately* after an incoming SMS message is received without the interaction from the user.

To construct unwanted behaviours from malware instances and benign apps, we generate sub-automata by calculating the intersection and difference between the behaviour automata of sample apps, and select the sub-automata which are strongly associated with and largely cover malware instances. Since this combinatorial construction and selection process is expensive, we adopt the approach proposed in [17] to accelerate it by exploiting the behavioural difference between malware instances and benign apps, and the family names of malware instances. This approach combines machine learning methods and text-mining techniques, and proceeds as follows.

1. Malware instances are organised into small groups according to their family names. Benign apps are added into each group to form a balanced training dataset.

2. For each group, we generate sub-automata by computing the intersection and difference between behaviour automata of apps in the same group, then train a linear classifier by taking these sub-automata as input features—checking whether a feature is a sub-automaton of the behaviour automaton of an app.

3. Those features which are actually used by the linear classifier are called *salient features*, i.e., their weights assigned by the linear classifier are not zero. We combine two groups by computing the intersection and difference between their salient features, then training a linear classifier on sample apps from these two groups to produce new salient features. This process continues until all groups are combined into a single group with a collection of salient sub-automata.

4. From these salient sub-automata an optimal subset is selected as *unwanted behaviours*. We apply text-mining techniques, e.g., subset-searching, weight ranking, and TF-IDF (term frequency - inverse document frequency) optimisation, etc., to help choose this subset, i.e., taking the salient features of the malware instances belonging to the same family as a document.

It took around two weeks to generate unwanted behaviours from apps in the training dataset using a multi-core desktop computer. We use the classification accuracy as the threshold to decide whether all features or only salient features are kept for each group. It was set to 90% in our implementation. At the end of computation, around 200 salient sub-automata are chosen as unwanted behaviours.

We list human-authored descriptions and learned unwanted behaviours for 10 prevalent families in Table 1. These descriptions for families were collected from their online analysis reports [2, 27, 34, 45]. A subjective comparison shows that unwanted behaviours compare well to human-authored descriptions. Also, they reveal the trigger conditions of some behaviours, which were often lacking in human-authored descriptions. For example, the expression BOOT_COMPLETED.SEND_SMS denotes that after the device finishes booting, this app will send a message out; the expression UMS_CONNECTED.LOAD_CLASS means that when a USB mass storage is connected to the device, this app will load some code from a library or a hidden payload; and the unwanted behaviour for Droiddream shows that if the phone state changes (the action PHONE_STATE), this app will collect some information then access Internet. In Table 1 only two behaviours are not captured by unwanted behaviours: "gain root access" for Droiddream and the behaviour of Spitmo.

Some behaviours of sample apps are not the same as unwanted behaviours, but, they often contain some unwanted behaviours as sub-sequences. For example, the behaviour SMS_RECEIVED.READ_PHONE_STATE.SEND_SMS contains the unwanted behaviour SMS_RECEIVED.SEND_SMS as a subsequence. To capture behaviours sharing the same patterns with the unwanted behaviours, if a behaviour contains an unwanted behaviour as a sub-sequence, we consider this behaviour as unwanted as well. We call them *extended unwanted behaviours*. For instance, we can generalise from the above unwanted behaviour and construct the automaton in Figure 2 as an extended unwanted behaviour. Here, we use the symbol Σ to denote the collection of events, actions, and permission-like phrases.

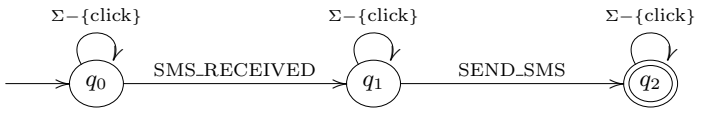

Figure 2: An example extended unwanted behaviour.

We check whether a target app has any unwanted behaviour ψ by testing whether $A \cap \psi = \emptyset$, where A is the behaviour automaton of the target app. These testing results will be used as input features in further training. Another usage of unwanted behaviours is to test whether $\psi \subseteq A$. This will result in high false negatives because A might not contain all unwanted behaviours specified in ψ.

3. GENERAL CLASSIFIERS

In this section, we will investigate robustness of general mobile malware classifiers. These classifiers were trained by applying supervised learning methods, including: decision trees [31], naive Bayes, L1-regularized linear regression [37], support vector machines [35], random forest [12], adaptive boosting [22], and k-nearest neighbours [5], and a semi-supervised learning method [15]. For each machine learning method, we trained using different syntax-based and semantics-based features which have been discussed in previous section, and tested on the validation and testing datasets which are described in Section 1. We will demonstrate that the best-performing features on the validation dataset, which are often syntax-based features, have poor classification performance on the testing dataset. We will show that semantics-based features dramatically improve the classification performance of the detection of new malware and achieve the best classification performance on the testing dataset for most machine learning methods we will compare.

The methods KNN (k-nearest neighbours), SVM (support vector machines) and NB (naive Bayes) are included in our study, because these methods have been successfully applied in the Android malware classification, e.g., DroidAPIMiner [3], Drebin [7], Yerima et.al. [41], etc. We will compare their classification performance, combine them with the semantics-based features and test on new malware.

The L1LR (L1-regularized linear regression) was deliberately designed to train classifiers on sparse data, i.e, only a small part of features is responsible for a decision. This assumption coincides with our intuition: features like API calls and happen-befores contain a lot of redundant information and most API calls or happen-befores are actually useless for the classification. Thus, we choose the L1LR as a candidate method to improve the classification performance.

The RF (random forest) is an ensemble learning method. It was designed to mitigate the overfitting problem in the DT (decision trees). Instead of training a single decision tree, it trains several trees respectively on random subsets of samples using random subsets of input features, and makes decisions by taking majority votings. This leads to a better model by decreasing the variance without increasing the bias, which is needed in our experience to obtain better robustness. Except for the RF, as a baseline, we include the DT in our comparison as well.

The AdaBoost (Adaptive Boosting) is another supervised learning method we have tested. It is an iterative process to produce stronger learners from weak learners. It improves the performance of a weak learner by adjusting the weights assigned to samples in favour of those misclassified by weak learners.

The SEMI (semi-supervised learning) is applied on a collection of labelled and unlabelled samples. It makes use of unlabelled samples for training to achieve a better classifier than doing supervised learning on the labelled samples or doing unsupervised learning on the unlabelled samples. This matches with our goal to detect new malware.

We use the tools *liblinear* [20] and *libsvm* [14] respectively to train L1LR and SVM classifiers. As for other methods, we use their implementations in *scikit-learn* [29]. We use the decision trees as the base estimators in the AdaBoost classifiers. For the semi-supervised learning, we adapt the model LabelSpreading to label unlabelled samples, which is an implementation of Zhou et al.'s work [44].

We report performance of general classifiers on the testing dataset in Figure 3. It shows that semantics-based features have better classification performance than syntax-based features. In particular, the best F_1-score of precision and recall is achieved by the classifier using unwanted behaviours and L1-Regularized Linear Regression. The precision and recall are calculated as follows:

$$\text{precision} = \frac{tp}{tp + fp} \qquad \text{and} \qquad \text{recall} = \frac{tp}{tp + fn}\,,$$

where tp, fp, and fn respectively denote the true positives, false positives, and false negatives.

The detailed classification performance is reported in Table 2. We summarise the main results as follows.

- *API calls achieve the best classification performance on the validation dataset.* The precision and recall of the classifiers using API calls as input features are respectively as high as 95% and 99%, e.g., in DT, RF, and SEMI classifiers.

- *The best-performing methods on the validation dataset are: DT, L1LR, RF, and SEMI, by using syntax-based features.* In particular, the average precision and recall for syntax-based features are respectively as high as 90% and 98%.

- *Syntax-based features have better classification performance on the validation dataset than semantics-based features.* The average precision and recall for syntax-based features on all tested methods are respectively 88% and 98%, while for semantics-based features these numbers are respectively 86% and 82%. That is, classifiers using syntax-based features have higher recall.

- *Syntax-based features perform badly on the testing data.* The average precision and recall on all tested methods are respectively 55% and 26%. In the worst case, the

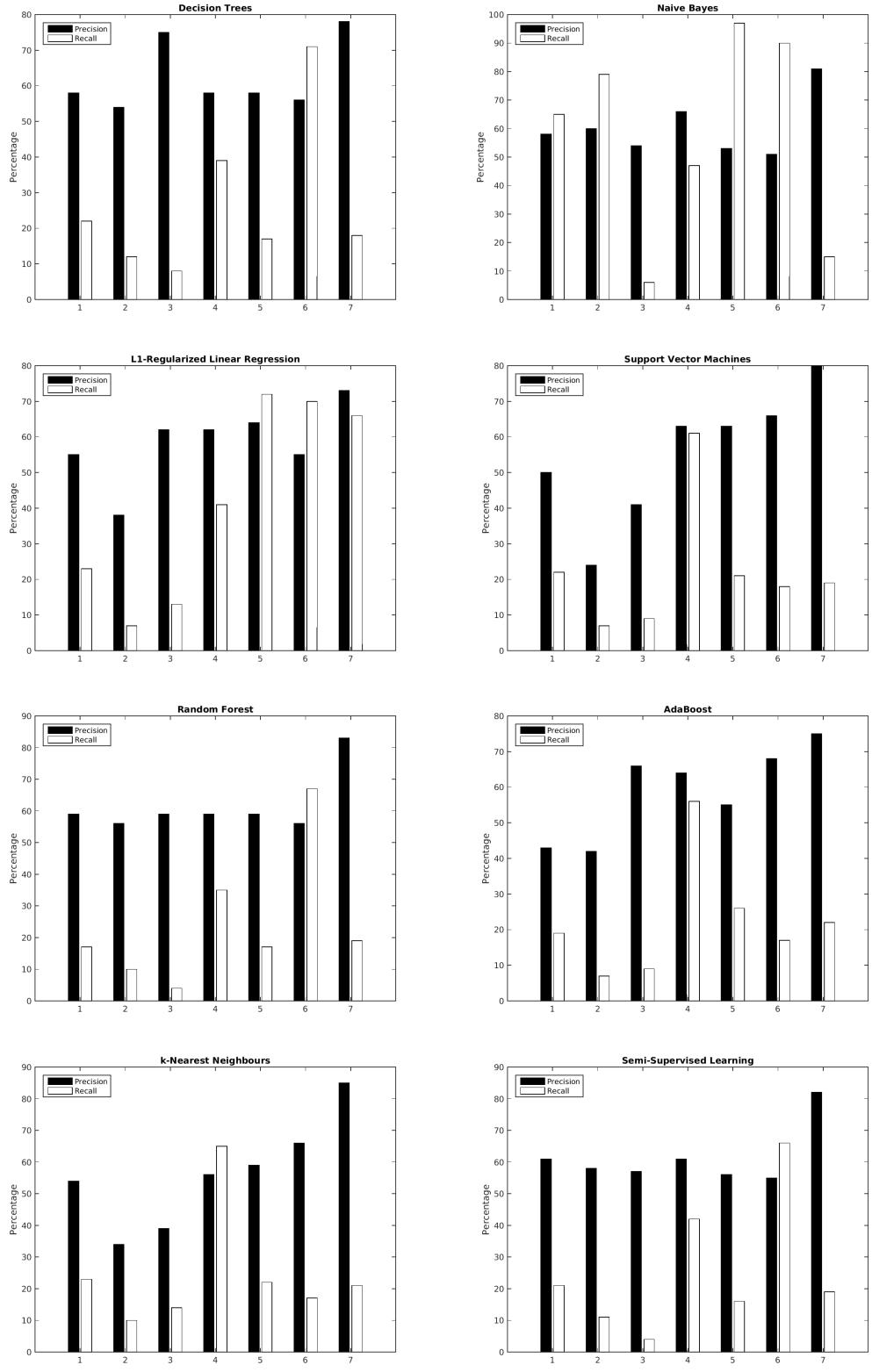

Figure 3: The classification performance of general classifiers on the testing dataset.

Decision Trees	*validation*		*testing*	
	precision	recall	precision	recall
signature-based features				
permissions	90%	99%	58%	22%
actions	91%	99%	54%	12%
API calls	95%	99%	75%	8%
keywords	86%	93%	58%	39%
average	91%	98%	61%	20%
semantics-based features				
reachables	93%	86%	58%	17%
happen-befores	68%	92%	**56**%	**71**%
unwanted	95%	73%	78%	18%
average	85%	84%	64% ↑	35% ↑

Naive Bayes	*validation*		*testing*	
	precision	recall	precision	recall
signature-based features				
permissions	74%	100%	58%	65%
actions	74%	99%	60%	79%
API calls	93%	99%	54%	6%
keywords	87%	91%	66%	47%
average	82%	97%	60%	49%
semantics-based features				
reachables	61%	99%	**53**%	**97**%
happen-befores	61%	98%	51%	90%
unwanted	96%	47%	81%	15%
average	73%	81%	62% ↑	67% ↑

L1-Regularized Linear Regression	*validation*		*testing*	
	precision	recall	precision	recall
signature-based features				
permissions	89%	99%	55%	23%
actions	90%	99%	38%	7%
API calls	93%	98%	62%	13%
keywords	88%	94%	62%	41%
average	90%	98%	54%	21%
semantics-based features				
reachables	73%	90%	64%	72%
happen-befores	68%	92%	55%	70%
unwanted	72%	72%	**73**%	**66**%
average	71%	85%	64% ↑	69% ↑

Support Vector Machines	*validation*		*testing*	
	precision	recall	precision	recall
signature-based features				
permissions	88%	99%	50%	22%
actions	88%	99%	24%	7%
API calls	91%	100%	41%	9%
keywords	82%	97%	**63**%	**61**%
average	87%	99%	45%	25%
semantics-based features				
reachables	93%	86%	63%	21%
happen-befores	93%	77%	66%	18%
unwanted	96%	71%	80%	19%
average	94%	78%	70% ↑	19% ↓

Random Forest	*validation*		*testing*	
	precision	recall	precision	recall
signature-based features				
permissions	91%	100%	59%	17%
actions	92%	99%	56%	10%
API calls	95%	99%	59%	4%
keywords	89%	92%	59%	35%
average	92%	98%	58%	17%
semantics-based features				
reachables	94%	87%	59%	17%
happen-befores	69%	92%	**56**%	**67**%
unwanted	96%	73%	83%	19%
average	86%	84%	66% ↑	34% ↑

AdaBoost	*validation*		*testing*	
	precision	recall	precision	recall
signature-based features				
permissions	87%	99%	43%	19%
actions	91%	99%	42%	7%
API calls	94%	99%	66%	9%
keywords	84%	94%	**64**%	**56**%
average	89%	98%	54%	23%
semantics-based features				
reachables	90%	89%	55%	26%
happen-befores	94%	78%	68%	17%
unwanted	94%	72%	75%	22%
average	93%	80%	66% ↑	22% ↓

***k*-Nearest Neighbours**	*validation*		*testing*	
	precision	recall	precision	recall
signature-based features				
permissions	89%	99%	54%	23%
actions	89%	99%	34%	10%
API calls	87%	99%	39%	14%
keywords	77%	97%	**56**%	**65**%
average	86%	99%	46%	31%
semantics-based features				
reachables	92%	85%	59%	22%
happen-befores	94%	77%	66%	17%
unwanted	95%	70%	85%	21%
average	94%	77%	70% ↑	20% ↓

Semi-Supervised Learning	*validation*		*testing*	
	precision	recall	precision	recall
signature-based features				
permissions	91%	100%	61%	21%
actions	91%	99%	58%	11%
API calls	95%	99%	57%	4%
keywords	87%	93%	61%	42%
average	91%	98%	59%	20%
semantics-based features				
reachables	94%	85%	56%	16%
happen-befores	69%	92%	**55**%	**66**%
unwanted	95%	72%	82%	19%
average	86%	83%	66% ↑	34% ↑

Table 2: The classification performance of general classifiers.

Training method	Training feature	ρ_1	$\rho_{0.5}$ ↓
NB	actions	76	71
L1LR	reachables	74	70
NB	reachables	72	70
L1LR	unwanted	71	70
NB	happen-befores	70	67
SVM	keywords	73	66
DT	happen-befores	70	65
AdaBoost	keywords	71	64
KNN	keywords	71	64
NB	permissions	71	64
L1LR	happen-befores	69	64
RF	happen-befores	69	64
SEMI	happen-befores	68	63
NB	keywords	68	59

Training method	Training feature	ρ_1	$\rho_{0.5}$ ↑
SEMI	API calls	14	9
RF	API calls	14	9
NB	API calls	19	13
SVM	actions	19	13
L1LR	actions	21	14
AdaBoost	actions	21	15
DT	API calls	25	17
SVM	API calls	26	18
KNN	actions	27	19
AdaBoost	API calls	27	19
RF	actions	29	20
SEMI	actions	31	22
DT	actions	33	23
L1LR	API calls	35	25

Table 3: The most and the least robust general classifiers.

precision and recall are respectively 24% and 7%, i.e., those of the SVM classifier using actions as input features; in the best case, these numbers are respectively 60% and 79%, i.e., those of the NB classifier using actions as input features.

- *Semantics-based features have better classification performance on the testing dataset than syntax-based features.* The average precision and recall for semantics-based features on all tested methods are respectively 67% and 38%. In the worst case, the precision and recall are respectively 56% and 16%, i.e., those of the SEMI classifier using reachables as input features; in the best case, these numbers are respectively 73% and 66%, i.e., those of the L1LR classifier using unwanted behaviours as input features.

- *The best-performing method on the testing dataset is L1LR, by using semantics-based features.* In particular, the average precision and recall for L1LR classifiers using the semantics-based features are respectively 64% and 69%.

- *Unwanted behaviours achieve the best classification performance on the testing dataset.* The L1LR classifier using unwanted behaviours as input features performs best on the testing dataset, in particular, the precision is 73% and the recall is 66%.

A *robust classifier* is required to perform well on the validation dataset as well as on the testing dataset. To achieve more robust classifiers, we want to pick up suitable features and machine learning methods according to the classification performance of 56 trained classifiers reported in Table 2. For this purpose, we introduce the following measure:

$$\begin{aligned} \rho_\beta &= (1+\beta^2)\frac{F_t \times F_v}{\beta^2 \times F_t + F_v} \\ F_t &= 2 \times \frac{P_t \times R_t}{P_t + R_t} \\ F_v &= 2 \times \frac{P_v \times R_v}{P_v + R_v} \end{aligned}$$

Here, the symbols P_t, R_t, P_v and R_v respectively denote the precision and recall of a classifier on the testing and the validation dataset. It is actually the F_β measure of two F_1-scores. The parameter β is usually set to 1 to get the harmonic mean of the classification performance on the testing and the validation datasets. By setting it to 0.5 we put more emphasis on the classification performance on the testing dataset. Table 3 displays the most robust 14 and the least robust 14 classifiers out of 56 general classifiers. We rank them by their $\rho_{0.5}$ values. From this table, we conclude:

- semantics-based features dramatically improve robustness of mobile malware classifiers;

- robustness of L1LR and NB classifiers is better than that of other classifiers.

This study reveals that syntax-based features usually lead to overfitting to training data. This is why their classification performance is good on the validation dataset but poor on the testing dataset. This drawback limits applications of classifiers trained using syntax-based features.

Since semantics-based features capture general behavioural invariants in malware, they can better characterise unwanted patterns in new samples. This leads to better classification performance on the testing dataset than syntax-based features. This success convinces us that semantics-based features have the potential to cope with zero-day malware.

4. CLUSTER-SPECIFIC CLASSIFIERS

A malicious behaviour in a group of mobile apps might be normal or innocuous in another group. For instance, sending SMS messages is normal for messaging apps, but unwanted for an E-reader app; accessing location is expected in a jogging tracer app but abnormal for a wallpaper app.

This observation motivates us to train using fine-grained groups of apps instead of the whole training dataset. The approach proceeds as follows.

1. Apps in the training dataset are orgranised into small groups by applying a clustering method. In our implementation, we use the method k-means [26] to cluster apps by computing the Euclidean distance between the binary vectors of features.

Clustering feature	Training method	Training feature	ρ_1	$\rho_{0.5}\downarrow$
reachables	L1LR	unwanted	74	72
-	NB	actions	76	71
keywords	L1LR	reachables	74	71
reachables	KNN	keywords	75	70
-	L1LR	reachables	74	70
happen-befores	L1LR	unwanted	72	70
-	NB	reachables	72	70
-	L1LR	unwanted	71	70
happen-befores	L1LR	reachables	73	69
unwanted	L1LR	reachables	73	69
reachables	L1LR	reachables	72	69
unwanted	L1LR	unwanted	70	69
keywords	KNN	keywords	73	68
reachables	NB	reachables	71	68
unwanted	KNN	keywords	72	67
keywords	L1LR	unwanted	70	67
happen-befores	NB	reachables	70	67
-	NB	happen-befores	70	67
-	SVM	keywords	73	66
happen-befores	KNN	keywords	72	66
keywords	SVM	keywords	72	66
reachables	SVM	keywords	72	65
happen-befores	SVM	keywords	72	65
-	DT	happen-befores	70	65
happen-befores	SEMI	happen-befores	69	65
happen-befores	RF	happen-befores	68	65

Table 4: The most robust general and cluster-specific classifiers.

2. We train a classifier for each group by using the machine learning methods and features which lead to the most robust general classifiers, e.g., L1-Regularised Linear Regression and unwanted behaviours, naive Bayes and reachables, decision trees and happen-befores, etc., so-called *cluster-specific classifiers*.

3. We select a group for each target app. In particular, we compute the Euclidean distance between the binary vectors of features and adopt the average-linkage [33] to measure the distance between a group and the target app. The closest group is chosen.

4. The cluster-specific classifier for the chosen group is applied to decide whether the target app is malware.

We trained 60 cluster-specific classifiers using top combinations of methods and features in Table 3. We evaluated their robustness and compared to that of general classifiers. The most robust (general and cluster-specific) classifiers are listed in Table 4. The detailed classification performance of cluster-specific classifiers is reported in appendix. We conclude:

- robustness of cluster-specific classifiers is better than general classifiers, especially, when the method L1LR, KNN, RF, AdaBoost, or SEMI is applied in training and semantics-based features are used for clustering;

- except for keywords, using syntax-based features for clustering will result in less robust cluster-specific classifiers than general classifiers;

- the most robust cluster-specific classifier is achieved by using the L1LR as the training method and semantics-based features for clustering and training.

By using semantics-based features in clustering, we organise apps based on their behaviours rather than signatures. This is why using semantics-based features to group apps leads to more robust classifiers than using syntax-based features. It confirms our intuition: an unwanted behaviour is a common behavioural pattern shared by malware within a group of apps which have similar behaviours.

5. CONCLUSION AND FURTHER WORK

We investigate robustness of machine-learning-based mobile malware classifiers. We apply supervised and semi-supervised learning methods, and extract syntax-based and semantics-based features to train general classifiers. By comparing the classification performance of these classifiers on the validation and testing datasets, we conclude: semantics-based features improve robustness of malware classifiers, in particular, it dramatically improves the classification performance on the testing dataset. A similar study on clustering-specific classifiers supports this argument as well.

However, semantics-based features might lead to underfitting to training data, i.e., their classification performance is not as good as syntax-based features on the validation dataset. A potential improvement is to add more fine-grained semantics-based features to achieve better fits to training data. Another is to combine syntax-based and semantics-based features in training. We will test these potential improvements in further work.

Extracting semantics-based features from apps is more expensive than extracting syntax-based features. It takes around 1 hour on average per app. But this effort is worthwhile. It will not only improve robustness of malware classifiers but also offer potential to understand and predict malicious behaviours in mobile apps.

In future, we want to further improve robustness of mobile malware classifiers by: (a) refining semantics-based features; (b) making use of the similarity between identified patterns and their variants in new unlabelled samples; (c) training and testing on market-scale datasets. It is also interesting to test the same argument on classifiers trained using the cutting-edge machine learning methods, e.g., deep learning [10, 42].

To efficiently learn unwanted behaviours from apps, we also want to develop a novel approach to combine machine learning methods and learning automata techniques [6, 11], such that semantics-based features can be applied in industry to obtain more robust classifiers over time.

6. REFERENCES

[1] Malware Genome Project. http://www.malgenomeproject.org/, 2012.

[2] Symantec security response. http://www.symantec.com/security_response/, 2015.

[3] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *SecureComm*, 2013.

[4] K. Allix et al. Are your training datasets yet relevant? In *ESSoS*, 2015.

[5] N. S. Altman. An introduction to kernel and Nearest-Neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[6] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[7] D. Arp et al. Drebin: Efficient and explainable detection of Android malware in your pocket. In *NDSS*, pages 23–26, 2014.

[8] S. Arzt et al. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, pages 259–269, 2014.

[9] K. W. Y. Au et al. PScout: Analyzing the Android permission specification. In *CCS*, pages 217–228, 2012.

[10] Y. Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, 2009.

[11] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.

[12] L. Breiman. Random forests. *Mach. Learn.*, 45, 2001.

[13] S. Chakradeo et al. Mast: Triage for market-scale mobile malware analysis. In *WiSec*, pages 13–24, 2013.

[14] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, 2011.

[15] O. Chapelle, B. Schlkopf, and A. Zien. *Semi-Supervised Learning*. The MIT Press, 2010.

[16] K. Z. Chen et al. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS*, 2013.

[17] W. Chen et al. On robust malware classifiers by verifying unwanted behaviours. In *12th International Conference on integrated Formal Methods*, 2016.

[18] L. Deshotels, V. Notani, and A. Lakhotia. DroidLegacy: Automated familial classification of Android malware. In *PPREW*, 2014.

[19] W. Enck et al. A study of Android application security. In *USENIX Security Symposium*, 2011.

[20] R.-E. Fan et al. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9, 2008.

[21] A. P. Felt et al. Android permissions demystified. In *CCS*, pages 627–638, 2011.

[22] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1), 1997.

[23] H. Gascon et al. Structural detection of Android malware using embedded call graphs. In *AISec*, pages 45–54, 2013.

[24] A. Gorla et al. Checking app behavior against app descriptions. In *ICSE*, 2014.

[25] J.-C. Kuester and A. Bauer. Monitoring real android malware. In *Runtime Verification 2015*, 2015.

[26] J. MacQueen. Some methods for classification and analysis of multivariate observations, 1967.

[27] McAfee Threat Center. http://www.mcafee.com/uk/threat-center.aspx, 2015.

[28] F. A. Narudin et al. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1):343–357, 2016.

[29] F. Pedregosa et al. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[30] H. Peng et al. Using probabilistic generative models for ranking risks of android apps. In *CCS*, pages 241–252. ACM, 2012.

[31] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[32] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 2nd edition, 1979.

[33] R. R. Sokal and C. D. Michener. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, 38:1409–1438, 1958.

[34] M. Spreitzenbarth et al. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153, 2015.

[35] I. Steinwart and A. Christmann. *Support Vector Machines*. Springer, 2008.

[36] G. Suarez-Tangil et al. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, 41(4, Part 1):1104 – 1117, 2014.

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

[38] F. Wei et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*, pages 1329–1341. ACM, 2014.

[39] C. Yang et al. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *ESORICS*, 2014.

[40] S. Yerima, S. Sezer, and I. Muttik. High accuracy android malware detection using ensemble learning. *IET Information Security*, 9(6), 2015.

[41] S. Y. Yerima et al. A new Android malware detection approach using Bayesian classification. In *AINA*, pages 121–128, 2013.

[42] Z. Yuan, Y. Lu, and Y. Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, Feb 2016.

[43] M. Zhang et al. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *CCS*, 2014.

[44] D. Zhou et al. Learning with local and global consistency. In *Advances in Neural Information Processing Systems 16*, pages 321–328, 2004.

[45] Y. Zhou and X. Jiang. Dissecting Android malware: characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

APPENDIX

A. THE CLASSIFICATION PERFORMANCE OF CLUSTER-SPECIFIC CLASSIFIERS

Decision Trees and Happen-Befores

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	68%	92%	**56**%	**71**%
permissions	84%	98%	38%	19%
actions	86%	99%	34%	12%
keywords	69%	91%	54%	69%
reachables	93%	86%	68%	20%
happen-befores	65%	91%	54%	72%
unwanted	95%	80%	75%	17%

Naive Bayes and Actions

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	74%	99%	**60**%	**79**%
permissions	85%	99%	63%	38%
actions	69%	100%	52%	73%
keywords	76%	99%	55%	63%
reachables	83%	100%	61%	54%
happen-befores	79%	100%	58%	57%
unwanted	79%	100%	54%	45%

Naive Bayes and Reachables

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	61%	99%	**53**%	**97**%
permissions	95%	85%	79%	20%
actions	80%	81%	45%	29%
keywords	90%	68%	60%	15%
reachables	66%	90%	57%	80%
happen-befores	64%	94%	53%	84%
unwanted	63%	89%	52%	79%

L1-Regularized Linear Regression and Unwanted

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	72%	72%	73%	66%
permissions	71%	99%	54%	55%
actions	73%	99%	59%	51%
keywords	64%	91%	54%	81%
reachables	74%	86%	**73**%	**67**%
happen-befores	74%	78%	74%	63%
unwanted	72%	71%	74%	64%

L1-Regularized Linear Regression and Reachables

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	73%	90%	64%	72%
permissions	71%	99%	56%	60%
actions	71%	99%	56%	55%
keywords	68%	93%	**65**%	**74**%
reachables	71%	88%	64%	69%
happen-befores	74%	86%	64%	70%
unwanted	74%	86%	69%	65%

Support Vector Machines and Keywords

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	82%	97%	63%	61%
permissions	86%	98%	47%	24%
actions	87%	99%	47%	20%
keywords	79%	91%	**61**%	**64**%
reachables	83%	92%	65%	57%
happen-befores	81%	93%	62%	60%
unwanted	84%	92%	52%	37%

Random Forest and Happen-Befores

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	69%	92%	56%	67%
permissions	83%	99%	45%	26%
actions	87%	99%	26%	8%
keywords	74%	90%	57%	64%
reachables	90%	87%	56%	23%
happen-befores	66%	84%	**56**%	**71**%
unwanted	95%	81%	77%	16%

AdaBoost and Keywords

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	84%	94%	**64**%	**56**%
permissions	86%	98%	55%	33%
actions	86%	99%	49%	24%
keywords	85%	87%	59%	40%
reachables	90%	93%	66%	32%
happen-befores	88%	90%	64%	41%
unwanted	85%	92%	63%	47%

k-Nearest Neighbours and Keywords

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	77%	97%	56%	65%
permissions	82%	99%	38%	23%
actions	85%	99%	51%	30%
keywords	73%	97%	**54**%	**83**%
reachables	73%	96%	61%	76%
happen-befores	74%	97%	56%	72%
unwanted	74%	96%	57%	73%

Semi-Supervised Learning and Happen-Befores

clustering	*validation*		*testing*	
feature	precision	recall	precision	recall
-	69%	92%	55%	66%
permissions	85%	98%	42%	19%
actions	87%	98%	44%	18%
keywords	73%	89%	54%	66%
reachables	93%	84%	70%	20%
happen-befores	67%	87%	**56**%	**71**%
unwanted	96%	80%	73%	16%