A Text-Mining Approach to Explain Unwanted Behaviours

A Text-Mining Approach to Explain Unwanted Behaviours

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

Current machine-learning-based malware detection seldom provides information about why an app is considered bad. We study the automatic explanation of unwanted behaviours in mobile malware, e.g., sending premium SMS messages. Our approach combines machine learning and text mining techniques to produce explanations in natural language. It selects keywords from features used in malware classifiers, and presents the sentences chosen from human-authored malware analysis reports by using these keywords. The explanation elaborates how a system decision was made. As far as we know, this is the first attempt to generate explanations in natural language by mining the reports written by human malware analysts, resulting in a scalable and entirely data-driven method.

CCS Concepts

•Computing methodologies → Learning paradigms; •Security and privacy → Mobile and wireless security; •Information systems → Data mining;

Keywords

Mobile security; Android system; malware detection; text mining; machine learning.

1. INTRODUCTION

Mobile malware, including trojans, spyware and other kinds of unwanted software, has been increasingly seen in the wild and even on official app stores [9, 26]. This has motivated recent research on dedicated methods for automatically identifying mobile malware, including methods based on static analysis [6, 11, 22, 23], dynamic analysis [8, 15, 19], and machine learning [4, 5, 13, 14, 24]. However, except some descriptions [20, 26] of several famous malware families, e.g., Geinimi, Basebridge, Droidkungfu, etc., people have no idea of which behaviour makes an app bad. It is not enough to simply identify an app as malware. We also need to give confidence that the identification is correct and explain what it means, so that appropriate actions are taken. This suggests an important research problem which has received far less attention: automatically generating a short paragraph to explain unwanted behaviours of an app if it has been automatically decided as malware.

Here are example automatic explanations for some instances in families: Zitmo, Opfake, and Droidkungfu [20, 26].

a. Allows applications to open network sockets, and uploads the data to a specific url. (an instance of Zitmo)
b. It sends SMS messages to premium rated numbers. (an instance of Opfake)
c. This is a trojan which steals personal information from the infected device. It can be controlled over the web through HTTP. (an instance of Droidkungfu)

This research has several potential benefits, including: producing hints for malware analysts before costly investigation, supporting the automatic generation of analysis reports, improving the understanding of threats in apps, etc.

Our approach combines machine learning and text mining techniques, and proceeds as follows.

- **Classifiers**. We train a linear classifier using permissions, actions, and API calls as input features. It not only automatically decides whether an app is malware but also characterises unwanted behaviours, i.e., a small set of features selected by their weights.
- **Keywords**. The selected features are converted into a set of keywords, i.e., natural language tokens extracted from the documents of these features.
- **Sentences**. We use these keywords to retrieve sentences from human-authored malware analysis reports.

If a target app is decided as malware by the classifier, the sentences for its features will be presented as the explanations of its unwanted behaviours.

EUROSEC'16,April 18-21, 2016, London, United Kingdom

http://dx.doi.org/10.1145/2905760.2905763

This is the first to automatically generate explanations in natural language by exploiting human-authored malware analysis reports. The main contributions follow.

- To generate explanations in natural language we leverage existing text, i.e., the text from the Android Developers [2] and malware analysis reports [20, 26].
- To keep the explanation compact and precise, the features and the keywords have to be carefully pruned. Simply using features greedily extracted from the classifier will lead to a lot of redundancy. For example, in the permission `SEND_SMS`, the API call `SmsManager.sendTextMessage`, and the action `SMS_RECEIVED`, the most informative keyword is "sms". Such redundancy wastes space that could have been used to add more information in the explanation. To combat this, we use the TF-IDF (term frequency - inverse document frequency) weighting and develop a new subset-search algorithm to choose the most informative keywords.

We evaluate our approach by comparing the automatic explanations to the manual descriptions, which were collected from malware analysis reports [20, 26] for around 200 malware families. We divided them into the training and testing sets, respectively for the sentence searching and the evaluation. We collected around $1,500$ malware instances across the families contained in the testing set and generated explanations for them. We measure the overlap between the keywords extracted from the automatic explanations and those extracted from the manual descriptions. This evaluation shows that the keyword-generation method doubles the precision of simply using features greedily extracted from classifiers and maintains approximately the same recall.

Drebin [5] was the first attempt to generate explanations for the mobile malware detection. It chooses the features with the top weights from an SVM classifier, and processes them through a set of hand-built templates to output text. The recent prototype DescribeMe [25], which generates text from data-flows, also uses hand-built templates. In our work, instead of building the templates by hand, we *automatically infer* the natural language templates, by leveraging the reports that are routinely written by malware analysts from anti-virus software vendors. We are unaware of previous work that exploits this rich source of information to generate explanations.

2. OVERVIEW

The main process of the automatic explanation is formalised as the function *explain*. It takes an app in question as input and produces a collection of sentences as the explanations.

We train a linear classifier W on a collection D of sample apps, which consists of $1,500$ malware instances and $1,500$ benign apps. The testing set also consists of $1,500$ malware instances and $1,500$ benign apps. It is disjoint with the training set. These apps were labelled and supplied by Intel Security. We adopt the L1-Regularized Logistic Regression [21] as the training method.

We collected around $41,000$ features, including: system permissions, actions, and API function names, from more than $10,000$ real apps. Their brief documents on the Android Developers were collected as well. From these documents we extracted keywords, which are nouns and technical terms, e.g., "sms", "gps", "url", etc. Keywords for each feature are ranked by their TF-IDF, then the top m keywords are chosen, i.e., K_U.

Function: $explain(app, U, M, D, m, n, \beta, \omega, \delta)$
Input: app – the application in question
U – a set of features and their documents
M – a training set of manual descriptions
D – a training set of applications
m – the maximum number of keywords per feature
n – the maximum number of features per application
β – the parameter for F_β measure
ω – the search width δ – the search depth
Output: the explanation of the target application

——— **Offline Training** ———

W : from features to weights (L1-Regularized Logistic Regression)
$W \leftarrow train(D)$

K_U: from features to keywords (TF-IDF)
$K_U \leftarrow keyword(U, m)$

S: from keywords to sentences (Cosine Similarity of TF-IDF vectors)
$S \leftarrow sentence(K_U, M)$

$K_D(a)$: keywords for an app a in D (top-n-negative)
$K_D \leftarrow select(D, n, W, K_U)$

——— **Online Explaining** ———

k_{app} : keywords for app
$k \subseteq k_{app}$: a subset of keywords for app

$E \leftarrow \emptyset$
$l \leftarrow$ benign
if $classify(W, app)$ is malware **then**
  $\{(app, k_{app})\} \leftarrow select(\{app\}, n, W, K_U)$
  Subset-Search for Keywords (in Section 5)
  $k \leftarrow search(k_{app}, K_D, \beta, \omega, \delta)$
  for w in k **do**
    $E \leftarrow E \cup \{S(w)\}$
  end for
  $l \leftarrow$ malware
end if
return (l, E)

We collected the manual descriptions from malware analysis reports [20, 26]. These manual descriptions were produced by malware analysts and third-party researchers. They were divided into the training and test sets. For each keyword in K_U, we search through descriptions contained in the training set for a central sentence including this keyword. This selection is based on the cosine similarity of TF-IDF vectors of sentences. This process gives a mapping S from keywords to sentences.

If the application in question is classified as malware by using W, we choose a maximum of n features from those with negative weights by ranking their absolute values, so-called *top-n-negative* (abbreviated as TNN). The sentences for these selected features are presented to explain its unwanted behaviours.

To improve the quality of generated keywords, in particular, the precision, we develop a subset-search algorithm, i.e., *search* in Section 5. It looks up a subset of keywords by exploring the difference between keywords extracted from malware instances and those from benign applications, such that it largely covers and is strongly associated with malware

instances. The parameters β, ω and δ are used to adjust the performance of this algorithm.

3. TRAINING A CLASSIFIER

From the manifest file of an Android app, we collect permissions requested by this app and actions registered in its intent-filters. Permissions reflect the requirement for resources. Actions are events which the app is interested in, triggered by the Android platform or other apps. For example, the permission `INTERNET` indicates that this app wants to use the Internet and the action `ACTION_ANSWER` denotes that this app can handle an incoming call. All API calls appearing in the code of an app are collected as well. The Android platform tools `aapt` and `dexdump` are used to help extract these features. We do not consider specific strings, e.g., IP addresses, URLs, etc., because they vary across different training sets.

We apply several popular machine learning methods to train classifiers. We are not only interested in the usual measures, e.g., accuracy and FPR (false positive ratio), but also especially in minimising the number of features which are actually used by classifiers, because a classifier that uses fewer features will be better suited for producing an explanation. We choose SVM, KNN, and naive Bayes which have been applied in Android malware detection respectively by Drebin [5], DroidAPIMiner [4], and Yerima et.al. [24]. We also compare to a decision tree classifier, because decision trees naturally employ a small number of features for each test instance, which could in principle be used to generate explanations, although as we will see later, these explanations turn out to be inferior.

We compare our target method L1-Regularized Logistic Regression (abbreviated as L1LR) [21] with the above methods. We use tools *liblinear* [10] and *libsvm* [7] respectively for L1LR and SVM. As for other methods, we use their implementations in *scikit-learn* [18].

We use the training and testing sets described in Section 2. The testing results are given as follows.

Method	Permission		Perm. & Action		Perm. & Action & API		
	Accuracy	FPR	Accuracy	FPR	Accuracy	FPR	Used Features
DT-C4.5	86.2%	14.6%	87.7%	13.6%	90.6%	9.5%	41,007
NB	81.4%	20.4%	85.8%	17.7%	87.7%	8.1%	41,007
KNN	86.8%	17.9%	86.4%	19.7%	86.8%	16.0%	41,007
SVM	86.4%	17.1%	88.4%	13.1%	90.6%	2.5%	15,140
L1LR	82.3%	26.0%	86.9%	17.7%	93.2%	7.0%	**2,265**

Only a small part of input features are actually useful to distinguish malware and benign apps, e.g., out of more than 41,000 input features, only 2,265 features are used in the L1LR-classifier. Notice that except for SVM and L1LR, the rest methods use all input features.

By adding new features, e.g., invoke-befores, denoting an API call is invoked before another, and trigger-befores, denoting an event is triggered before an API call, the classification accuracy of L1LR can be further improved to around 98% and the FPR can be reduced to around 2%. However, since our goal is trying to understand unwanted behaviours of malware, rather than incrementally obtaining better fits to a training dataset, we prefer to keep our current choice of input features: permissions, actions, and API calls.

4. SELECTING FEATURES

We want to identify a small set of features that were responsible for the classification decision, to characterise unwanted behaviours. Intuitively, for a linear classifier, a feature with a negative weight more likely indicates an unexpected behaviour, and a feature with a positive weight more likely indicates a normal behaviour.

Based on this observation, we want to choose the best method from the following: from the features of the app in question, (a) select all features; (b) select all features with negative weights; (c) randomly select n features from those with negative weights; (d) select top n features from those with negative weights, ranked by their absolute values; (e) use features appearing on a path from the root to a leaf in a decision tree. Among them, (b), (c), and (d) work for features used in SVM or L1LR classifiers. The method (e) only works for decision trees.

We measure how well keywords extracted from selected features match with those extracted from the manual descriptions, i.e.,

$$\text{precision} = \frac{|K_g \cap K_f|}{|K_g|} \qquad \text{and} \qquad \text{recall} = \frac{|K_g \cap K_f|}{|K_f|},$$

where K_g is the collection of generated keywords for a malware instance and K_f is the collection of keywords extracted from the manual descriptions of its family.

By keywords, we denote the remaining words after removing meaningless words, stop-words, verbs, adjectives, and adverbs. For instance, keywords for the text "this application turns an Android smartphone into a GPS tracker", are "gps" and "tracker". Here, the words "application", "Android", and "smartphone" are considered as stop-words. For each feature, we also extract keywords from its name, e.g., keywords for the action `Telephony.SMS_RECEIVED` are "telephony" and "sms".

Feature Selection Method	Precision	Recall
all-features (a)	0.091	**0.362**
DT-C4.5-path (e)	0.028	0.002
L1LR-all-negative (b)	0.109	0.295
SVM-all-negative (b)	0.100	0.331
L1LR-random-5-negative (c)	0.131	0.094
SVM-random-5-negative (c)	0.119	0.070
The method used in Drebin [5]		
SVM-top-5-negative (d)	0.295	0.182
The target method		
L1LR-top-5-negative (d)	**0.327**	**0.184**

In the above table, we give the evaluation results of different methods to select features. We test on the testing set described in Section 2. The method (d) achieves the best precision and recall. It confirms that choosing features by ranking their weights is more effective than using all features or randomly selecting features with negative weights. The upper bound of recall is around 36%. This reflects that the language used in specifying features and the language used in composing manual descriptions are very different.

To understand the change of precision and recall on the parameter n for methods (c) and (d), we apply them on features used in L1LR and SVM classifiers respectively, with n ranging from 1 to 100. We depict the results in Figure 1. It shows that when the number of selected features increases, the precision decreases very quickly while the recall doesn't increase much. That means more and more redundancy is introduced into generated keywords when more negative fea-

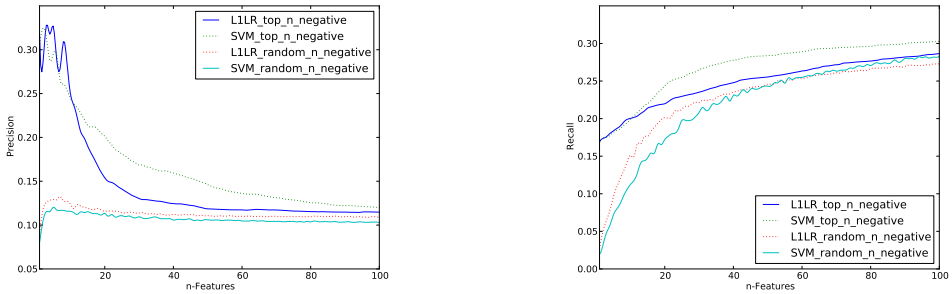

Figure 1: Top-n-negative versus random-n-negative. The precision decreases dramatically when the number of selected features increases.

tures with smaller absolute values of weights are added. It confirms that with respect to those features with negative weights, the bigger the absolute value of a feature's weight is, the more likely this feature indicates an unwanted behaviour.

Based on the above discussions, we choose top-n-negative to select features from those used by a linear classifier.

5. GENERATING KEYWORDS

To improve the quality of generated keywords, in particular, to boost the precision, we want to choose a subset X of keywords such that it largely covers and is strongly associated with malware instances. Formally, we write $P_D(X)$ and $R_D(X)$ to respectively denote the probability of an app is malware if it has all keywords from X and the probability of an app has all keywords from X if it is malware, where D is a collection of malware instances and benign apps. We adopt F_β-measure of them as the evaluation function, i.e.,

$$F_\beta(X, D) = (1 + \beta^2) \cdot \frac{P_D(X) \cdot R_D(X)}{\beta^2 \cdot P_D(X) + R_D(X)}.$$

To exhaustively search the space of the power-set of keywords is expensive. Based on the Beam Search [17, Chapter 6] we design an algorithm to approximate the best subset with size at most δ. It is formalised as the following.

Function: $search(k, K_D, \beta, \omega, \delta)$
Input: k – a set of keywords
K_D – keywords for background dataset D
β – evaluation parameter
ω – search width δ – search depth
Output: an approximation of the best subset of keywords

$p \leftarrow [\,]$; $q \leftarrow [\,]$; $r \leftarrow [\,]$ {working max-priority-queues}
for t in k **do**
  $enqueue(q, (\{t\}, F_\beta(\{t\}, K_D)))$
end for
$q \leftarrow dequeue(q, \omega)$ {control the search width}
$r \leftarrow dequeue(q, 1)$ {get the best singleton}
while q is not empty **do**
  $(l, e) \leftarrow dequeue(q, 1)$
  if $size(l) > \delta$ **then**
    continue {control the search depth}
  end if
  for t in k **do**
    if t is not in l **then**
      $enqueue(p, (l \cup \{t\}, F_\beta(l \cup \{t\}, K_D)))$
      $enqueue(r, (l \cup \{t\}, F_\beta(l \cup \{t\}, K_D)))$
    end if
  end for
  if q is empty **then**
    $q \leftarrow dequeue(p, \omega)$ {add ω-best successors}
    $p \leftarrow [\,]$
    $r \leftarrow dequeue(r, 1)$ {record the best subset}
  end if
end while
$(s, \_) \leftarrow dequeue(r, 1)$ {get the best subset}
return s

In our implementation, we set the search width ω to 10 and the search depth δ to 5. The parameter β in F_β-measure is set to 0.5 since we are more concerned with how largely they covers malware instances.

Instead of extracting keywords directly from the names of features, we extract keywords from their documents. For example, the brief document for the action `ACTION_ANSWER` is "Activity Action: Handle an incoming phone call". We choose keywords for each feature by ranking them using TF-IDF. These documents provide more informative keywords, e.g., "incoming" and "call", than the feature names.

By subset-searching TF-IDF ranked keywords, the precision increases from around 30% to around 60%, compared with only using the top-n-negative. This is shown in Figure 2. For L1LR the best precision is achieved when the parameter n is set to 8. The recall is maintained at around 20%.

Keywords generated by using different methods are reported in Table 1. Each row of this table denotes the selected features and keywords of a malware instance.

No.	Top-n-Negative (TNN)	TNN & TF-IDF	TNN & TF-IDF & subset-search
1	SEND_SMS SmsManager.sendTextMessage FileChannel.force READ_PHONE_STATE action.USER_PRESENT	sms text channels system	sms
2	NetworkInfo.getSubtype WebSettings.getUserAgentString READ_PHONE_STATE TelephonyManager.getLine1Number http.entity.HttpEntityWrapper	information http	information
3	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION WAKE_LOCK VIBRATE INTERNET	location internet	location
4	action.USER_PRESENT action.BOOT_COMPLETED Telephony.SMS_RECEIVED conn.CONNECTIVITY_CHANGE	device sms provider connectivity	device

Table 1: Salient features and keywords for malware.

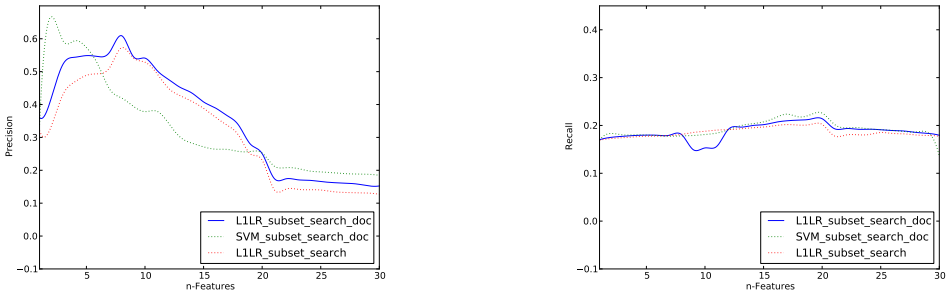

Figure 2: Subset-search TF-IDF ranked keywords. It doubles the precision and maintains the recall.

6. SEARCHING FOR SENTENCES

Keywords give some useful and concise information. But, they are not so interesting to attract attentions. For example, it is much better to present the sentence "It steals information and uploads the stolen data to a remote server". than giving the keyword "information".

In our approach, we use keywords to look up sentences from manual descriptions of malware families, resulting in a natural language explanation for a malware instance. For each keyword, from the sentences including this keyword, we select a single *central sentence*, i.e., the sentence that is most similar to all the others. We adopt the cosine similarity of TF-IDF vectors of sentences as the evaluation function to rank sentences. The details are as follows. Let C be the set of all sentences containing a word w. For each sentence s in C, we first construct its TF-IDF vector $V[s]$ as:

$$V[s][a] = \begin{cases} tfidf(a, s, C), & \text{if } a \text{ is a word in } s; \\ 0, & \text{otherwise.} \end{cases}$$

for all words a appearing in C. Then, we use the following sum of cosine similarity between TF-IDF vectors:

$$\sigma_V(s) = \sum_{t \in C} cos(\theta_{V[s],V[t]}) = \sum_{t \in C} \frac{V[s] \times V[t]}{\|V[s]\| \cdot \|V[t]\|}$$

as the measure of a sentence s. The sentence in C having the highest value is chosen.

Some example automatic explanations of malware instances and the manual descriptions of their malware families are given in Table 2. These examples are randomly selected from the testing set. This comparison qualitatively shows that the automatic explanations compare well to the manual descriptions.

7. CONCLUSION AND FURTHER WORK

We have presented a new text-mining approach to generate natural language explanations of unwanted behaviours of Android apps. Ours is the first method to leverage previously written malware descriptions from anti-virus vendors in order to generate natural text for new malware instances. In contrast, most previous work on malware detection focuses on obtaining good fits to a given collection of sample apps by trying different methods and features [4, 12, 24]. Explanations of features which are responsible for the classification decision have received much less consideration.

The purpose of training and comparing classifiers in this paper is to demonstrate that the weights assigned by linear classifiers can help figure out indicative features. In practice, our method can be extended to take any well-trained linear classifier's weights as input.

We evaluate our method by measuring the overlap between keywords extracted from automatic explanations and those extracted from malware analysis reports produced by human malware analysts. This evaluation shows that the keyword generation method doubles the precision of simply using features greedily extracted from classifiers. Measuring the quality of generated sentences is difficult. To give a qualitative evaluation of automatic explanations, in further work, we want to survey end users and malware analysts to obtain the most convincing automatic explanations.

The method to select central sentences from manual descriptions is simple. In further work, we want to investigate and apply more complex sentence synthesis techniques to produce explanations, especially, to improve the recall of generated sentences.

There are still certain types of behaviours exhibited in Android malware but cannot be fully captured by our approach, e.g., gaining root access [26], performing DDoS attacks [1], intercepting incoming messages, etc. This is because these behaviours do not correspond to single features in the classifiers. In further work, a promising approach to remove this limitation might be to exploit more semantics-based features, e.g., commands, modules, call graphs, subsequences of API call traces, etc., to capture these behaviours. This will lead to more accurate explanations.

On the other hand, the unwanted behaviours for a group of apps might be normal for another. For example, people are happy with a Jogging Tracer app accessing the locations but uncomfortable with an E-Reader app doing so. So, in further work, we want to investigate whether the information like categories, family names, and clusters can help further improve automatic explanations.

For zero-day malware and unknown unwanted behaviours, this supervised-learning and text-mining approach will not work. To produce explanations by combining semi-supervised

Malware Family	Automatic Explanation	Manual Description
BaseBridge	It sends **sms** messages to premium rated numbers. This is a Trojan which steals personal **information** from the infected device.	"A Trojan horse that attempts to send premium-rate SMS messages to predetermined numbers." from [3]. "Forwards confidential details (SMS, IMSI, IMEI) to a remote server." from [1].
Plankton	It can be controlled over the web through **http**. This is a Trojan which steals personal **information** from the infected device, and uploads the data to a specific **url**.	"A Trojan that steals sensitive information. The Trojan attempts to send gathered information to a remote machine." from [3]. "This malware has the capabilities to communicate with a remote server, download and install other applications, send premium rated SMS messages, and many many more..." from [1].
DroidKungfu	This is a Trojan which steals personal **information** from the infected device. It can be controlled over the web through **http**.	"A Trojan that sends sensitive information to an attacker and includes backdoor functionality. It also exploits vulnerabilities to gain root access." from [16]. "Collects a variety of information on the infected phone (IMEI, device, OS version, etc.). The collected information is dumped to a local file which is sent to a remote server afterwards" from [1].
JiFake	It sends **sms** messages to premium rated numbers. This is a Trojan which steals personal **information** from the infected device.	"This application sends premium rated SMS messages." from [1]. "Sends SMS messages to a premium rate number." from [16].

Table 2: Automatic explanations versus manual descriptions. The keywords are shown in boldface.

learning methods and sentence synthesis techniques might be worth exploring.

8. REFERENCES

[1] Forensic Blog. http://forensics.spreitzenbarth.de/android-malware/, 2014.

[2] Android Developers. http://developer.android.com/index.html, 2015.

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

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

[5] D. Arp, M. Spreitzenbarth, M. HÃijbner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of Android malware in your pocket. *NDSS*, pages 23–26, 2014.

[6] 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.

[7] 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, May 2011.

[8] W. Enck et al. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, 2014.

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

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

[11] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *SIGSOFT FSE*, 2014.

[12] M. Frank, B. Dong, A. P. Felt, and D. Song. Mining permission request patterns from Android and Facebook applications. In *ICDM*, pages 870–875, 2012.

[13] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *AISec*, pages 45–54, 2013.

[14] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *ICSE*, 2014.

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

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

[17] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers Inc., 1st edition, 1992.

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

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

[20] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp, and J. Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153, 2015.

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

[22] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*, pages 1329–1341. ACM, 2014.

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

[24] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new Android malware detection approach using bayesian classification. In *AINA*, pages 121–128, 2013.

[25] M. Zhang, Y. Duan, Q. Feng, and H. Yin. Towards automatic generation of security-centric descriptions for Android apps. In *CCS '15*, pages 518–529, 2015.

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