

Explaining Unwanted Behaviours in Context

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

David Aspinall
University of Edinburgh
David.Aspinall@ed.ac.uk

Andrew D. Gordon
Microsoft Research Cambridge
University of Edinburgh
Andy.Gordon@ed.ac.uk

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

Igor Muttik
Intel Security
igor.muttik@intel.com

Abstract

Mobile malware has been increasingly identified based on unwanted behaviours like sending premium SMS messages. However, unwanted behaviours for a group of apps can be normal for another, i.e., they are context-sensitive. We develop an approach to automatically explain unwanted behaviours in context and evaluate the automatic explanations via a user-study with favourable results. These explanations not only state whether an app is malware but also elaborate how and in what kind of context a decision was made.

1 Introduction

Researchers and malware analysts have identified hundreds and thousands of mobile apps as malware [EOMC11, ZJ12] and organised them into families based on some unwanted behaviours, e.g., stealing personal information, accessing locations, collecting contacts information, sending premium messages constantly, etc. However, except for some malware analysis reports of several famous malware families [ZJ12, S⁺13], e.g., Geinimi, Basebridge, Spitmo, Zitmo, Ginmaster, Ggtracker, Droidkungfu, etc., people don't know what kind of behaviour makes a mobile app bad. This suggests a research problem: automati-

cally producing a short paragraph to explain unwanted behaviours.

A naive method is to: train a linear classifier from a collection of identified malware instances and benign apps, choose features with top weights assigned by this classifier, then process selected features through templates to output text. This method has been adopted in research, e.g., the Drebin system [A⁺14a].

However, by greedily choosing features to output text, the generated explanations are inaccurate. This is mainly because unwanted behaviours of mobile apps are *context-sensitive*, i.e., an unwanted behaviour in one group of apps can be normal in another. For example, collecting locations is normal for jogging tracker apps, but unwanted for card game apps.

Instead, our new approach is to organise sample apps into fine-grained groups by their behavioural similarity. We set the context of an app in question to the group whose members' behaviours are the most similar to this app's behaviours. By exploiting behavioural difference between malware and benign apps in this context, we decide whether the target app is malware, and if so, we produce an explanation. Here are two example automatic explanations.

a. This app is like a chatting app, but, after a USB massive storage is connected, it will: retrieve a class in a runnable package; read information about networks; connect to Internet.

b. This app is like an anti-virus app, but, it will: read your phone state after a phone call is made; read your phone state then connect to Internet; send SMS messages after a phone call is made.

These explanations not only elaborate which behaviour is unwanted but also give the context, e.g., chatting and anti-virus, in which a decision was made. Here, the context name is from the known category names of apps in the same group by majority voting.

Copyright © by the paper's authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

In: D. Aspinall, L. Cavallaro, M. N. Seghir, M. Volkamer (eds.): Proceedings of the Workshop on Innovations in Mobile Privacy and Security IMPS at ESSoS'16, London, UK, 06-April-2016, published at <http://ceur-ws.org>

Our approach combines static analysis, clustering, supervised-learning, and text mining techniques, and proceeds as follows.

- **Formalisation.** We approximate the behaviour of an Android app by an extended call-graph, i.e., a collection of finite control-sequences of events, actions, and annotated API calls. From this graph we extract the *happen-before* feature which denotes that something happens before another.
- **Learning.** We organise sample apps into groups using clustering methods, and characterise unwanted behaviours for each group by exploring the difference between malware instances and benign apps within the same group.
- **Explanation.** We decide whether a target app is malware by choosing a group then checking against this group whether the app has any unwanted behaviour, i.e., a behaviour exhibited by a malware instance in the group. The corresponding features are fed through hand-built templates to produce text as explanations.

The main contributions of this paper are to:

- show that the happen-before feature is an appropriate abstract of app behaviours with respect to learning and explaining;
- introduce the context into behaviour explanations and develop a clustering-based algorithm (Figure 2) to organise sample apps into groups and construct unwanted behaviours for each group;
- demonstrate that the automatic explanation in context produces more convincing and desirable results than several other candidate methods by surveying general users.

1.1 Related Work

To automatically detect Android malware, machine learning methods have been applied to train classifiers [ADY13, GYAR13, GTGZ14, YSMM13]. All of them were to obtain good fits to the training data by trying different methods and features. Explanations of chosen features have received much less consideration.

The tool Drebin [A⁺14a] is the first attempt to automatically generate explanations of Android malware. It generates explanations by choosing features with top weights from a linear classifier then processes them through hand-built templates to output text. A broad range of syntax-based features, e.g., permissions, API calls, intents, URLs, etc., were collected for training.

A recent prototype DescribeMe [ZDFY15] generates text from data-flows by feeding features through hand-built templates. The main drawback is its scalability: to produce data-flows is too expensive for most apps.

The idea of context is similar with the cluster used in the tool CHABADA [GTGZ14]. This tool detects the outliers (abnormal API usage) within the clusters of apps by using OC-SVM (one-class SVM). These clusters were grouped by the descriptions of apps using LDA (Latent Dirichlet Allocation). However, for most sample apps which were collected from alternative Android markets, e.g., Wandoujia, Baidu, and Tencent in China, it is hard to get their descriptions and these descriptions are often written in different languages.

The extended call-graphs are much more accurate than the manifest information, e.g., permissions and actions, which were often used as input features for malware detection or mitigation [BKvOS10, EOM09, F⁺11]. Compared with a simple list of API calls appearing in the code, the extended call-graph can capture more sophisticated behaviours. This is needed in practice, because: API calls appearing in the code contain “noise” caused by the dead code and libraries [ADY13]; and, some unwanted behaviours only arise when some API methods are called in certain orders [C⁺13, KB15, Y⁺14]. On the other hand, call-graphs are less accurate than models which capture data-flows. But, it is much easier to generate the extended call-graphs using our tool for apps en masse than generating data-flows using tools like FlowDroid [A⁺14b] or Amandroid [WROR14]. In particular, people can annotate appealing API methods to generate compact graphs more efficiently, rather than considering all data-dependence between statements.

2 Characterising App Behaviours

We use a simplified synthetic example to illustrate the characterisation of app behaviours. It is an Android app which constantly sends out the device ID and the phone number by SMS messages in the background when an incoming SMS message is received.

We approximate its behaviour by using the graph in Figure 1. It tells us: this app has two entries which are respectively specified by actions MAIN and SMS_RECEIVED; it will collect the device ID and the phone number in a Broadcast Receiver, then send SMS messages out in an AsyncTask; the behaviour of sending SMS messages can also be triggered by an interaction from the user, e.g., clicking a button, touching the screen, long-pressing a picture, etc., which is denoted by the word “click”.

This graph is a collection of finite control-sequences of actions, events, and annotated API calls, which is constructed from the bytecode of an Android app. Ac-

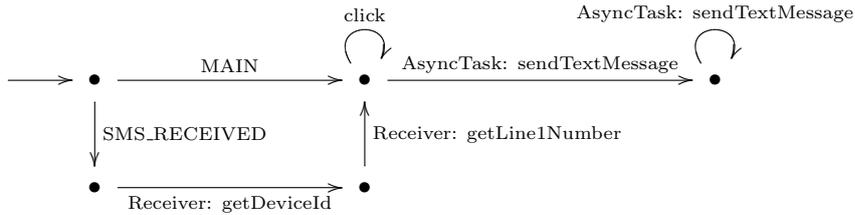


Figure 1: An example extended call-graph.

tions reflect what happens in the environment and what kind of service an app requests for, e.g., an incoming message is received, the device finishes booting, the app wants to send an email by using the service supplied by an email-client, etc. Events denote the interaction from the user, e.g., clicking a picture, pressing a button, scrolling down the screen, etc. Annotated API calls tell us whether the app does anything we are interested in. For instance, `getDeviceId`, `getLine1Number`, and `sendMessage` are annotated API calls in the above example.

To construct such a graph directly from the bytecode, we have to model complex real-world features of the Android framework, including: inter-procedural calls, callbacks, component life-cycles, permissions, actions, events, inter-component communications, multiple threads, multiple entries, interfaces, nested classes, and runtime-registered listeners. We don’t model registers, fields, assignments, operators, pointer-aliases, arrays or exceptions. The choice of which aspect 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 [AZHL12] as annotations. The Android platform tools `aapt` and `dexdump` are respectively used to extract the manifest information and to decompile the bytecode into the assembly code, from which we construct the extended call-graph.

Once the extended call-graphs are constructed, we can extract features for the purpose of learning unwanted behaviours. In particular, we extract pairs of edge labels occurring in sequence, i.e., denoting that something happens before another, so-called *happens-before*. Generally, one can extract n -tuples. But, in practice, we found that constructing triples was already too expensive: the order of magnitude for the average number of triples in a typical extended call-graph is 10^4 .

3 Learning Unwanted Behaviours

A behaviour that is unwanted for one kind of app can be innocuous for another. For example, sending SMS messages is normal for messaging apps, but unwanted for an E-reader app; a user might expect that

a weather forecast app accesses his or her locations, but might feel uncomfortable if a messaging app does so. Therefore, to understand and explain unwanted behaviour, we need a notion of context.

3.1 Constructing Context

Unwanted behaviours in general only account for a small part of a malicious app’s activities. This is by design: malicious apps seek to hide their bad behaviours, and are often constructed by repackaging benign applications [Z⁺14, Z⁺13]. This observation gives us a notion of context: we group together apps, benign or malicious, whose behaviours are mostly the same. Then, within the context, we distinguish unwanted from normal behaviours by exploring features which are mostly associated with malware. This produces a fine-grained, behavioural notion of context, that is more discriminating than categories, e.g., GAME, TOOLS, and WEATHER, etc., or clusters produced from developer-written textual descriptions [GTGZ14].

We formalise this idea in Figure 2. Sample apps are organised into groups. Apps in the same group share common behaviours, in the sense that their feature vectors are similar. Ideally, repackaged apps will be in the same group with the original benign apps. In practice, a group might consist of only benign apps or only malware. This depends on the feature used for clustering and its distribution in sample apps.

Two sets of features are constructed for each group: normal and unwanted. The normal set is the union of all behaviours of benign apps. The unwanted set consists of abnormal behaviours of malware, that is, the relative complement of the normal set in the collection of behaviours of malware instances.

The rule behind this construction is: *a benign app can not have any unwanted behaviour and a malware instance must have some unwanted behaviour whatever its other behaviours are*. Every sample app in the same group is required to follow this rule. Otherwise, there is a conflict in the group. To solve this conflict, we split the group into two disjoint subgroups. Then, the above construction will be done respectively on subgroups until all conflicts are solved.

```

Function construct_context (group)
Input: a group of malware and benign applications
Output: fine-grained groups with normal and unwanted features.
G ← {group}
P ← {}
has_conflict ← True
while has_conflict do
  has_conflict ← False
  for group in G do
    normal, unwanted ← collect_behaviour (group)
    if detect_conflict (group, normal, unwanted) then
      group_a, group_b ← split_group (group)
      G = (G - {group}) ∪ {group_a, group_b}
      has_conflict ← True
    else
      G = G - {group}
      P = P ∪ {(group, normal, unwanted)}
    end if
  end for
end while
return P

Function collect_behaviour (group)
normal ← {}
unwanted ← {}
for app in group do
  if app is benign then
    normal = normal ∪ feature(app)
    unwanted = unwanted - normal
  else
    unwanted = (unwanted ∪ feature(app)) - normal
  end if
end for
return normal, unwanted

Function detect_conflict (group, normal, unwanted)
for app in group do
  if app is benign and feature(app) ⊈ normal then
    return True
  end if
  if app is malicious and feature(app) ∩ unwanted = ∅ then
    return True
  end if
end for
return False

```

Figure 2: Context and unwanted behaviours.

The process starts in the function *construct_context* which is invoked on the whole collection of sample apps. When the algorithm terminates the following property is satisfied: for each app in a group, if it is malware then $feature(app) \cap unwanted \neq \emptyset$; if it is benign then $feature(app) \subseteq normal$.

The function *split_group* splits a group of apps into two disjoint subgroups. Many implementations are possible. We adopt the hierarchical clustering method to group apps. The cosine dissimilarity between feature vectors is calculated and the average-linkage is used to calculate the distances between clusters.

To illustrate the notion of context we constructed unwanted behaviours of 400 randomly-chosen sample apps by using the above method. The ten biggest generated groups are given in Table 1.

3.2 Classification

We want to decide whether an app in question is malware, by using the constructed context and unwanted behaviours. The size and the portion of malware vary largely across groups, as shown in Table 1. This re-

sults in: it is hard to train a classifier for each group using classical learning methods, e.g., SVM, naive Bayes, and logistic linear regression. Therefore, we calculate the distances between the target app and each group. The closest group is chosen as the context. Then, we decide whether the target app is malware by applying the following logic rules:

- **Conservatively normal.** The target app is classified as benign if it has no unwanted behaviour and all its behaviours are normal, i.e., $feature(app) \subseteq normal$.
- **Aggressively malicious.** The target app is classified as malicious if one of its behaviours is unwanted, i.e., $feature(app) \cap unwanted \neq \emptyset$.
- **Neutrally suspicious.** If the target app has no unwanted behaviour but some of its behaviours are not normal. We consider its abnormal behaviours, i.e., $feature(app) - normal$, as suspicious and label it as unknown. That is, according to current knowledge we can not decide whether it is malware. The decision have to be postponed until more sample apps of this group are acquired.

We randomly chose 1,000 apps with benign and malicious half-and-half as the training set; and an equal number of apps as the testing set. They contain some famous benign apps, i.e., Google Talk, Amazon Kindle, Youtube, Facebook, etc., and some instances in famous malware families, e.g., DroidKungfu, Plankton, Zitmo, etc. These apps spread in around 30 categories from ARCADE GAME to WEATHER. Many advertisement libraries were also found in these apps, e.g., Admob, Millennial Media, Airpush, etc.

To compare our classification method with general classifiers, we train a classifier using an implementation *liblinear* [F⁺08] of L1-Regularized Logistic Regression [Tib94] (abbreviated as L1LR). We apply our method to construct context and collect unwanted behaviours from happen-befores which are extracted from the extended call-graphs of apps in the training set. Further, we apply the logic rules discussed earlier, to decide whether a target app is malware against unwanted behaviours for a chosen group.

We report the classification performance as follows.

Classifier	Edge Labels in Graphs		Happen-Befores	
	Precision	Recall	Precision	Recall
context	83%	88%	80%	92%
L1LR	83%	89%	85%	88%

It shows that for different features the classification performance of our method is only slightly worse than L1LR, with no more than a 5% drop in precision. This is because some apps are labelled as unknown in our method. We can achieve better classification

Group	Size	%Malware	#Normal	#Unwanted	Top Malware Family	Top Category
0	163	93.25	6825	36813	Geinimi Fakerun	ENTERTAINMENT PERSONALIZATION
5	24	100.0	0	2611	Basebridge Spitmo	COMMUNICATION MUSIC_AND_AUDIO
21	23	39.13	3306	734	Plankton Droidkungfu	WEATHER PHOTOGRAPHY
7	17	41.18	1466	295	unknown	WEATHER TRANSPORTATION
19	13	15.38	1396	77	Adrd	COMMUNICATION TOOLS
12	10	10.0	2027	39	Adrd	MUSIC_AND_AUDIO NEWS_AND_MAGAZINES
25	8	0.0	227	441	-	WEATHER BOOKS_AND_REFERENCE
4	7	85.71	497	584	unknown	GAME_STRATEGY
15	7	85.71	20	2907	unknown	TRAVEL_AND_LOCAL WEATHER
6	5	40.0	764	125	unknown	PRODUCTIVITY NEWS_AND_MAGAZINES

Table 1: Statistics of context and unwanted behaviours for 400 sample apps.

performance by adding syntax-based features, e.g., permissions and API calls, as input features. However, our goal is to develop a classification method whose output yields better explanations. Considering happen-befores can capture more sophisticated app behaviours, we prefer to using unwanted behaviours selected from happen-befores for the explanation generation.

4 Generating Explanations

In the classification against the context, the features in the intersection between unwanted behaviours of a context and behaviours of a target app are responsible for a decision, so-called *salient features*. For a training app in a decision context, if one of its behaviours is salient, then this app is a *supporting app* for this decision. In this section, we want to exploit salient features and their supporting apps to generate an explanation for a target app. It explains how and in what kind of context a decision was made. We want to use these automatic explanations to convince people of the system’s automatic decision. Here is an example automatic explanation.

com.keji.danti590 (v3.0.8)

This application is malware. Its malicious behaviours are:

```
after a USB mass storage is connected,

it gets the superclass of a class in a runnable package
it retrieves classes in a runnable package
it reads information about networks
it connects to Internet
it reads your phone state then connects to Internet
```

The supporting apps of this explanation are:

```
com.keji.danti607 (v3.0.8) (TROJAN)
com.jjdd (v1.3.1) (MALWARE)
com.keji.danti562 (v3.0.8) (TROJAN)
com.keji.danti599 (v3.0.8) (TROJAN)
```

It not only shows the decision (malware or benign) but also elaborates the most unwanted behaviours. A collection of supporting apps is displayed as well.

Before presenting technical details, let us have a look at some salient features:

- a. (Object:ConnectivityManager.getActiveNetworkInfo, Runnable:URL.openConnection)
- b. (Activity:WifiManager.isWifiEnabled, Activity:WebView.loadUrl)
- c. (Object:WebView.loadUrl, Runnable:WifiInfo.getMacAddress)
- d. (AsyncTask:DefaultHttpClient.execute, Runnable:URL.openConnection)
- e. (Object:WebView.loadData, Runnable:TelephonyManager.getDeviceId)
- f. (AsyncTask:NotificationManager.notify, Object:LocationManager.getLastKnownLocation)

They are pairs extracted from the extended call-graphs of the apps in question. Some of them are trivial, e.g., the behaviour “access networks state then connect to Internet”, supported by the feature (Object:ConnectivityManager.getActiveNetworkInfo, Runnable:URL.openConnection), appears in almost every app. Some of them are similar, e.g., if we want to capture the behaviour “connect to Internet”, then features URLConnection.openConnection and DefaultHttpClient.execute are considered as repeated features. This redundancy will further clutter the final explanation.

Based on these observations, we generate explanations as follows: map these salient features into simple phrases, process simple phrases through templates to output compound phrases, then select the most representative compound phrases to present.

First, for each permission, action, event, and each API call which is not governed by any permission, a phrase is assigned to describe its function. These phrases were extracted from their brief documents on Android Developers. Second, for those permission-governed API calls, we look up their corresponding permissions and use phrases for these permissions. Third, for pair features we combine phrases for their coordinates to form compound phrases. The templates used in explanation are listed in Table 2. This step actually aggregates features to reduce redundancy.

By using the above method, for each supporting app, we get a collection of phrases with their appearance frequencies in this app. We rank phrases for each supporting app using the TF-IDF (term frequency - inverse document frequency) and choose the top-*m* phrases as representatives. Then, we apply DF (document frequency) to rank representatives of supporting apps and choose the top-*n* phrases to present. We use

Feature Type	Template	Example
permission	request the permission to <i>do sth.</i>	request the permission to change Wi-Fi connectivity state
API call	might invoke the API: <i>API name</i>	might invoke the API: android.content.Intent.<init>
annotation	<i>do sth.</i>	read your phone state
action	<i>sth. happens</i>	the app has finished booting
event	the user <i>does sth.</i>	the user clicks a view and holds
(annotation, annotation)	<i>do sth. then do sth.</i>	read your phone state then connect to Internet
(annotation, action)	<i>do sth. then sth. happens</i>	read SMS then the app makes a phone call
(action, annotation)	after <i>sth. happens do sth.</i>	after the system has finished booting read your phone state
(event, annotation)	when the user <i>does sth. do sth.</i>	when the user touches the screen get your precise location
(event, action)	when the user <i>does sth. sth. happens</i>	when the user performs a gesture the app sends some data to someone elsewhere

Table 2: Templates for the explanation generation.

formulae $\left(0.5 + \frac{0.5 \times f(t,d)}{\max\{f(t,d)|t \in d\}}\right) \times \log_{10} \frac{|C|}{|\{d|t \in d\}|}$ and $\log_{10} \frac{|\{d|t \in d\}|}{|C|}$ to respectively calculate TF-IDF and DF, where d is the collection of phrases for each app, C is the collection of all d , and $f(t, d)$ denotes the appearance frequency of t in d . This step helps remove trivial phrases (features), and is formalised as follows.

Function *gen_exp* (app, judge, group, normal, unwanted, m, n)

Input: the target app, the decision context, and the control parameters m and n .

Output: the explanation of the target app.

```

salient ← {}
if judge is malicious then
  salient ← feature(app) ∩ unwanted
else
  salient ← feature(app) ∩ normal
end if
supp ← {}
corpus ← {}
for app in group do
  features ← feature(app) ∩ salient
  if features ≠ ∅ then
    for feature in features do
      phrase ← feature_to_phrase(feature)
      if not phrase in doc then
        doc[phrase] ← 0
      end if
      doc[phrase] ← doc[phrase] + frequency(feature, app)
    end for
    supp ← supp ∪ {app}
    corpus ← corpus ∪ {(app, doc)}
  end if
end for
exp ← sel_df(sel_tfidf(corpus, m), n)
return judge, exp, supp

```

The function *feature_to_phrase* constructs a phrase for a given feature by using templates given in Table 2. Functions *sel_tfidf* and *sel_df* will respectively select phrases for each supporting app and representatives for the whole collection of supporting app. The function *frequency* produces the frequency of a feature appearing in an app.

5 Evaluation

In this section, we report a user-evaluation of the automatic explanations. We want to show: (a) expla-

nations produced from semantics-based features are better than from syntax-based features; (b) explanations with supporting apps are more understandable than without; (c) explanations produced from context construction are more convincing and preferable than greedily extracting features from general classifiers. To test these hypotheses, we design and compare the following methods.

- **M-Syntax:** By applying the context construction, from the *syntax*-based features (permissions and API calls), we produce explanations *without including* supporting apps.
- **M-Semantics:** By applying the context construction, from the *semantics*-based features (happen-befores), we produce explanations *without including* supporting apps.
- **M-Context:** By applying the context construction, from the *semantics*-based features (happen-befores), we produce explanations *including* supporting apps.
- **M-L1LR:** By using features with top weights in an L1LR classifier, which is trained from the *semantics*-based features (happen-befores), we produce explanations *including* supporting apps.

We applied the above methods to generate explanations for apps in the testing set which has been described in Section 3.2. The generated explanations were organised into samples. Each sample consists of two explanations for the same app, which are respectively produced by applying two different methods. Two example samples are given in Figure 3.

We chose three or four samples for each hypothesis testing. A survey consisting of 12 samples covering 10 malware instances and 2 benign apps was presented to participants. Participants are invited to read through

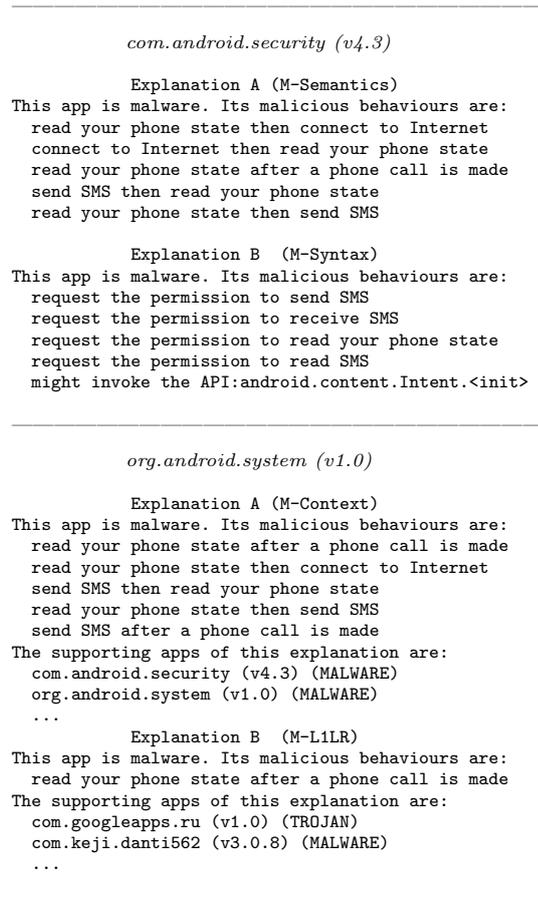


Figure 3: Example explanations for hypothesis testing.

all samples and for each sample, to choose the explanation which they prefer, and to give a convince-score between 1 and 5 to each explanation. This score indicates to what extent an explanation convinces the participant. We collected participants’ preferences as well as convince-scores.

People from universities, software companies, and finance firms in UK and China were invited by mailing lists to participate in this survey. All participants have no idea of the mechanism behind the automatic explanation discussed in this paper. We received 20 responses. These respondents include: seven junior and one senior software engineers, seven postgraduate students, one lecturer, three data analysts, and one malware analyst. Three of them declared to be familiar with Android programming and malware analysis.

We report the user-evaluation results as follows.

Method	Convince-score		Comparison	Preference
	Average	Std.		
M-Syntax	3.15	0.85	M-Context	58%
M-Semantics	3.03	0.66	M-Syntax	42%
M-Context	3.61	0.80	M-Context	78%
			M-Semantics	22%
M-L1LR	3.32	0.81	M-Context	53%
			M-L1LR	47%

It shows that the context construction achieves the highest average convince-score 3.61 and most respondents prefer explanations produced by the context construction. We do paired T-test respectively on the three comparisons: **M-Context** versus **M-Syntax**, **M-Context** versus **M-Semantics**, and **M-Context** versus **M-L1LR**. We set the significance level at 0.05, then calculate the difference between their convince-scores and test the null hypothesis: the average is less than or equal to 0. Their p-values are 0.02, 0.0002, and 0.05 respectively. That is, all null hypotheses are rejected at significance level 0.05. The automatic explanation by applying the context construction is better than alternative methods.

Respondents commented that explanations revealed some behaviours they had not realised before, e.g., an app called “com.antivirus.kav” sends SMS after a phone call is made, and supporting apps improve their understanding of the given explanation, e.g., they prefer to believing the given explanation is benign when they see familiar benign app names like Google Talk in the supporting apps. But, some of them, especially the malware analyst and those postgraduate students, wanted to see detailed features we use to produce explanations. This explains why **M-Syntax** is slightly better than **M-Semantics** in this surveying: API names are included in explanations produced by **M-Syntax** but not in **M-Semantics**. In practice, we can hide detailed features from users and only present them on-demand as evidence.

6 Conclusion and Further Work

We present a new approach to automatically generate explanations of unwanted behaviours of Android apps. It exploits semantics-based features, constructs context-sensitive unwanted behaviours, and produces explanations by aggregating features into phrases.

The context we have constructed is simple and straightforward. As shown in Table 1, the groups are unbalanced—some of them consists of hundreds of apps and some consists of several malware instances. In further work, we want to construct more balanced and fine-grained groups such that the supervised learning methods can be applied to obtain well-performing classifiers. By doing so, our approach to generate explanations can be extended to take features from well-trained classifiers as input.

A good classifier might not lead to a good explainer. As shown in Section 5, the explanations produced using the method **M-L1LR** are not the most preferable, although the L1LR classifier has better classification performance. To evaluate the quality of the automatic explanations is difficult. In this paper, we surveyed 20 general users to show the effectiveness of our method.

In further work, instead of general users, we want to survey a bigger group of malware analysts, since malware analysts are more suitable readers of these explanations. Also, more complex statistical models like ANOVA will be applied to analyse surveying results.

There are still certain types of high-level behaviours that are exhibited in Android malware but cannot be fully captured by our approach, e.g., gain root access and perform DDoS attacks [ZJ12]. This is because these complex behaviours do not correspond to simple semantics-based features like happen-befores. In further work, a promising approach to remove this limitation might be to exploit more semantics-based features to capture these high-level behaviours.

References

- [A⁺14a] Daniel Arp et al. Drebin: Efficient and explainable detection of Android malware in your pocket. *NDSS*, pages 23–26, 2014.
- [A⁺14b] Steven Arzt et al. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, pages 259–269, 2014.
- [ADY13] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *SecureComm*, 2013.
- [AZHL12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android permission specification. In *CCS*, 2012.
- [BKvOS10] David Barrera, Hilmi Günes Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *CCS*, 2010.
- [C⁺13] Kevin Zhijie Chen et al. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS*, 2013.
- [EOM09] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. On lightweight mobile phone application certification. In *CCS*, pages 235–245, 2009.
- [EOMC11] William Enck, Damien Ocateau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *USENIX Security Symposium*, 2011.
- [F⁺08] Rong-En Fan et al. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008.
- [F⁺11] Adrienne Porter Felt et al. Android permissions demystified. In *CCS*, 2011.
- [GTGZ14] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *ICSE*, 2014.
- [GYAR13] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of Android malware using embedded call graphs. In *AISec*, pages 45–54, 2013.
- [KB15] Jan-Christoph Kuester and Andreas Bauer. Monitoring real android malware. In *Runtime Verification 2015*, 2015.
- [S⁺13] Michael Spreitzenbarth et al. Mobile-sandbox: Having a deeper look into Android applications. In *SAC*, 2013.
- [Tib94] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.
- [WROR14] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Aandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*, 2014.
- [Y⁺14] Chao Yang et al. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *ESORICS*, 2014.
- [YSMM13] Suleiman Y. Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. A new Android malware detection approach using bayesian classification. In *AINA*, 2013.
- [Z⁺13] Wu Zhou et al. Fast, scalable detection of "piggybacked" mobile applications. In *CODASPY '13*, 2013.
- [Z⁺14] Fangfang Zhang et al. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *WiSec*, 2014.
- [ZDFY15] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. Towards automatic generation of security-centric descriptions for Android apps. In *CCS*, 2015.
- [ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.