Explaining Unwanted Behaviours in Context

Citation for published version:

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
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  
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 some malware analysis reports for several famous malware families [ZJ12, S+13], e.g., Geinimi, Basebridge, Spitmo, Zitmo, Ginnmaster, Ggtracker, Droidkungfu, etc., people don’t know what kind of behaviour makes a mobile app bad. This suggests a research problem: automatically 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 which are 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 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 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.

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 a finite-state automaton, i.e., a collection of finite control-sequences of events, actions, and annotated API calls. From this automaton we extract *happen-before* features to denote 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:

- demonstrate 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 method to organise sample apps into groups;
- remove redundancy in features by carefully choosing phrases to present;
- compare our approach with several alternative methods and report favourable evaluation results based on surveying end 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. However, the syntax-based features cannot capture sophisticated behaviours; greedily choosing features with top weights might produce inaccurate explanations; the selected features include redundancy which will clutter the final explanation.

Our approach overcomes these limitations by: using semantics-based features, e.g., *happen-befores*; introducing the context and collecting context-sensitive unwanted behaviours; aggregating and selecting phrases to present.

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, not mentioning that these descriptions are in different languages.

Automata 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, an automaton 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, automata are less accurate than models which capture data-flows. But, it is much easier to generate automata 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 behaviour automata 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 automaton 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”. All states in this automaton are accepting states since any prefix of an app’s behaviours is one of its behaviours as well.

This automaton is a collection of finite control-sequences of actions, events, and annotated API calls, which is constructed from the bytecode of an Android app. Actions 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 application 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 sendTextMessage are annotated API calls in the above example.

To construct such an automaton 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 features 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 automaton.

Once automata 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 happen-befores. 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 automaton 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 hat their feature vectors are similar. Ideally, repackaged apps will be in the same group with the original benign apps. A group can consist of only benign apps or only malware. Two sets of features are constructed: 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

![Figure 1: An example behaviour automaton.](image-url)
Function \texttt{construct\_context}(\texttt{group})

Input: a group of malware and benign applications

Output: fine-grained groups with normal and unwanted features.

\begin{verbatim}
G ← \{\}
P ← \{
has\_\text{conflict} ← \text{True}
\end{verbatim}

while has\_\text{conflict} do

\begin{verbatim}
for group in G do

normal, unwanted ← \text{collect\_\text{behaviour}}(group)

if \text{detect\_\text{conflict}}(group, normal, unwanted) then

G = (G − \{group\}) ∪ \{\text{group}_a, \text{group}_b\}

has\_\text{conflict} ← \text{True}

else

G = G − \{group\}

P = P ∪ \{\{group, normal, unwanted\}\}

end if
\end{verbatim}

end for

end while

return P

Function \texttt{collect\_\text{behaviour}}(\texttt{group})

\begin{verbatim}
normal ← \{
unwanted ← \{
\end{verbatim}

for app in group do

\begin{verbatim}
if \text{app is benign} then

normal = normal \cup \text{feature}(app)

unwanted = unwanted − normal

else

unwanted = (unwanted \cup \text{feature}(app)) − normal

end if
\end{verbatim}

end for

return \texttt{normal, unwanted}

Function \texttt{detect\_\text{conflict}}(\texttt{group}, normal, unwanted)

\begin{verbatim}
for app in group do

if \text{app is benign and feature}(app) \not\subseteq normal then

return \text{True}

end if

if \text{app is malicious and feature}(app) \cap unwanted = \emptyset then

return \text{True}

end if

end for

return \text{False}
\end{verbatim}

Figure 2: Context and unwanted behaviours.

the group into two disjoint subgroups. Then, the above construction will be done respectively on subgroups until all conflicts are solved.

The process starts in the function \texttt{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 \text{feature}(app) \cap unwanted \neq \emptyset; if it is benign then \text{feature}(app) \subseteq normal.

The function \texttt{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.

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. This results 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., \text{feature}(app) \subseteq normal.

- **Aggressively malicious.** The target app is classified as malicious if one of its behaviours is unwanted, i.e., \text{feature}(app) \cap unwanted \neq \emptyset.

- **Neutrally suspicious.** If the target app has no malicious behaviour and some of its behaviours are abnormal. We consider its abnormal behaviours 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, e.g., 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 \textit{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 automata 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.

<table>
<thead>
<tr>
<th>classifier</th>
<th>edge labels in automata</th>
<th>happen-befores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>precision</td>
<td>recall</td>
</tr>
<tr>
<td>context</td>
<td>83%</td>
<td>88%</td>
</tr>
<tr>
<td>L1LR</td>
<td>83%</td>
<td>89%</td>
</tr>
</tbody>
</table>

We report the classification performance as above. 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 performance by adding syntax-based features,
e.g., permissions and API calls, as input features. Rather, 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:

- 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.danti607 (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. (Object:LocationManager.getLastKnownLocation, Activity:WifiInfo.getMacAddress)
- e. (AsyncTask:DefaultHttpClient.execute, Runnable:URL.openConnection)
- f. (Object:WebView.loadData, Runnable:URL.openConnection)
- g. (AsyncTask:NotificationManager.notify, Object:LocationManager.getLastKnownLocation)
- h. (Click, Object:TelephonyManager.getDeviceId)
- i. (Object:ConnectivityManager.getActiveNetworkInfo, Object:DefaultHttpClient.execute)

They are pairs extracted from the automata 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 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 1. 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 the following formulae 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).

We formalise the above approach in Figure 3. The function \( \text{feature_to_phrase} \) constructs a phrase for a given feature by using templates given in Table 1. Functions \( \text{sel}_\text{tfidf} \) and \( \text{sel}_\text{df} \) will respectively select phrases for each supporting app and representatives for the whole collection of supporting app. The function \( \text{frequency} \) 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) explanations produced from semantics-based features are better than those produced from template-based features.
<table>
<thead>
<tr>
<th>feature type</th>
<th>template</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>permission</td>
<td>request the permission to do sth.</td>
<td>request the permission to change Wi-Fi connectivity state</td>
</tr>
<tr>
<td>API call</td>
<td>might invoke the API: API name</td>
<td>might invoke the API: android.content.Intent.&lt;init&gt;</td>
</tr>
<tr>
<td>annotation</td>
<td>do sth.</td>
<td>read your phone state</td>
</tr>
<tr>
<td>action</td>
<td>sth. happens</td>
<td>the app has finished booting</td>
</tr>
<tr>
<td>event</td>
<td>the user does sth.</td>
<td>the user clicks a view and holds</td>
</tr>
<tr>
<td>(annotation, annotation)</td>
<td>do sth. then do sth.</td>
<td>read your phone state then connect to Internet</td>
</tr>
<tr>
<td>(annotation, action)</td>
<td>do sth. then sth. happens</td>
<td>read SMS then the app makes a phone call</td>
</tr>
<tr>
<td>(action, annotation)</td>
<td>after sth. happens do sth.</td>
<td>after the system has finished booting read your phone state</td>
</tr>
<tr>
<td>(event, annotation)</td>
<td>when the user does sth. do sth.</td>
<td>when the user touches the screen get your precise location</td>
</tr>
<tr>
<td>(event, action)</td>
<td>when the user does sth. sth. happens</td>
<td>the app sends some data to someone elsewhere</td>
</tr>
</tbody>
</table>

Table 1: Templates for the explanation generation.

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

if features ∉ φ then
for feature in features do
phrase ← feature doe_phrase(feature)
doc[phrase] ← 0
end if
end for

for app in group do
features ← feature(app) ∩ salient
if features ∉ φ then
for feature in features do
phrase ← feature doe_phrase(feature)
doc[phrase] ← frequency(feature, app)
doc[phrase] ← doc[phrase] + frequency(feature, app)
end if
supp ← supp ∪ {app}
corpus ← corpus ∪ {(app, doc)}
end if
end for

exp ← select([sel IDF(sel IDF(corpus, m), n)]
return judge, exp, supp

Figure 3: Generating explanations.

Table: Comparison of explanation methods with supporting apps.

<table>
<thead>
<tr>
<th>Method</th>
<th>Explanation</th>
<th>Support App</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-Syntax</td>
<td>(happen-befores)</td>
<td>without including supporting apps.</td>
</tr>
<tr>
<td>M-L1LR</td>
<td>(happen-befores)</td>
<td>including supporting apps.</td>
</tr>
<tr>
<td>M-Context</td>
<td>(happen-befores)</td>
<td>without including supporting apps.</td>
</tr>
<tr>
<td>M-L1LR</td>
<td>(happen-befores)</td>
<td>including supporting apps.</td>
</tr>
</tbody>
</table>

Twelve apps are randomly chosen from the test set. We apply the above methods to generate explanations for these apps. The generated explanations are 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 4.

For each sample, a participant of this survey was invited to choose the explanation which he/she prefers, 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.

We use the Google Form to create this survey. 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 collected data from the first 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. Two of them haven’t used any Android app.
6 Conclusion

We have presented a new approach to automatically generate explanations of unwanted behaviours for mobile apps. A user-study by surveying end users shows that this approach is effective. 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


