Learning and Verifying Unwanted Behaviours

Wei Chen[1], David Aspinall[1], Andrew D. Gordon[1,2], Charles Sutton[1], and Igor Muttik[3]

[1] University of Edinburgh, UK,
{wchen2,csutton}@inf.ed.ac.uk,
{David.Aspinall,Andy.Gordon}@ed.ac.uk
[2] Microsoft Research Cambridge, UK,
[3] Intel Security, UK,
igor.muttik@intel.com

Abstract. Unwanted behaviours, such as interception and forwarding of incoming messages, have been repeatedly seen in Android malware. We study the problem of learning unwanted behaviours from malware instances and verifying the application in question to deny these behaviours. We approximate an application's behaviours by an automaton, i.e., finite control-sequences of events, actions, and annotated API calls, and develop an efficient machine-learning-centred method to construct and choose abstract sub-automata, to characterise unwanted behaviours exhibited in hundreds and thousands of malware instances. By taking the verification results against unwanted behaviours as input features, we show that the performance of detecting new malware is improved dramatically, in particular, the precision and recall are respectively 8% and 51% better than those using API calls and permissions, which are the best performing features known so far. This is the first automatic approach to generate unwanted behaviours for machine-learning-based Android malware detection. We also demonstrate unwanted behaviours constructed for well-known malware families. They compare well to those described in human-authorised descriptions of these families.

Keywords: mobile security, static analysis, software verification, machine learning, malware detection

1 Introduction

Android malware, including trojans, spyware and other kinds of unwanted software, has been increasingly seen in the wild and even on official app stores [2, 4, 22, 45]. Researchers and malware analysts have organised malware instances into hundreds of families [37, 45], e.g., Basebridge, Geinimi, Ginmaster, Spitmo, Zitmo, etc. These malware instances share certain unwanted behaviours, for example, sending premium messages constantly, collecting personal information, loading classes from hidden payloads then executing commands from remote servers, and so on. Except some inaccurate online analysis reports [1, 3, 5, 6, 32] of identified malware families, however, people have no idea of what exactly happens in these malware instances.

We want to learn unwanted behaviours from hundreds and thousands of malware instances and verify the application in question to deny them. We will show that these unwanted behaviours can improve the classification performance of detecting new malware. Our approach is outlined as follows.

- **Formalisation**. We approximate an Android application's behaviours by a finite-state automaton, that is, finite control-sequences of events, actions, and annotated API calls. Since different API calls might indicate the same behaviour, we abstract the automaton by aggregating API calls into permission-like phrases. We call it a *behaviour automaton*.
- **Learning**. An *unwanted behaviour* is a common behaviour which is shared by malware instances and has been rarely seen in benign applications. We develop a machine-learning-centred method to infer unwanted behaviours, by efficiently constructing and selecting sub-automata from behaviour automata of malware instances. This process is guided by the behavioural difference between malware and benign applications.
- **Refinement**. To purify unwanted behaviours, we exploit the family names of malware instances to help figure out the most informative unwanted behaviours. We compare unwanted behaviours with the human-authorised descriptions for malware families, to ensure that they match well with patterns described in these descriptions.
- **Verification**. We check whether the application in question has any security fault by verifying whether the intersection between its behaviour automaton and an unwanted behaviour is not empty.

To simulate new malware detection, we take malware instances released in different years and collected from different sources respectively as training, validation and testing sets. We use a group of malware and benign applications released before 2014 as the training and validation sets. Malware instances in these sets were collected from Malware Genome Project [45] and Mobile-Sandbox [37]. We take a collection of malware and benign applications released in 2014 as the testing set. They were supplied by Intel Security. We use API calls, permissions, and the verification results against unwanted behaviours as input features; then apply L1-Regularized Linear Regression [30, 39] to train classifiers. The evaluation on the testing set shows that the precision and recall of using unwanted behaviours are respectively 8% and 51% better than those of using API calls and permissions, which are the best performing input features known so far for machine-learning-based Android malware detectors. As shown in Table 2, using API calls and permissions as input features, can achieve very good precision and recall on the validation set, however, its classification performance on the testing set is poor. That is, unwanted behaviours are more general than API calls and permissions. This is needed in practice, to mitigate over-fitting and improve the robustness of malware classifiers.

Our approach is the first to automatically generate unwanted behaviours for Android malware classification. The main contributions of this paper are:

- We show that it is hard to detect new malware for classifiers trained on identified malware. We demonstrate that by using semantics-based features

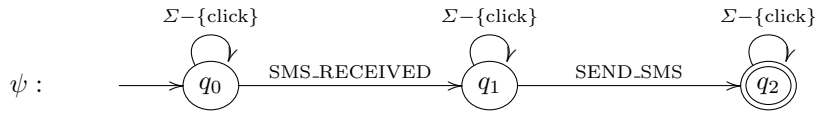
like unwanted behaviours, the classification performance of detecting new malware is dramatically improved.
- We have designed and implemented a static analysis tool to construct behaviour automata from the byte-code of Android applications, with regard to a broad range of features of the Android framework.
- The complexity of constructing all sub-automata is exponential in the number of malware instances. We propose and apply a new machine-learning-centred algorithm to combat this.
- We develop a refinement approach to look up the most informative unwanted behaviours, by making use of the family names of malware instances.

2 An Example Unwanted Behaviour

Let us consider a malware family called Ggtracker. A brief description of this family, which was produced by Symantec [6], is as follows.

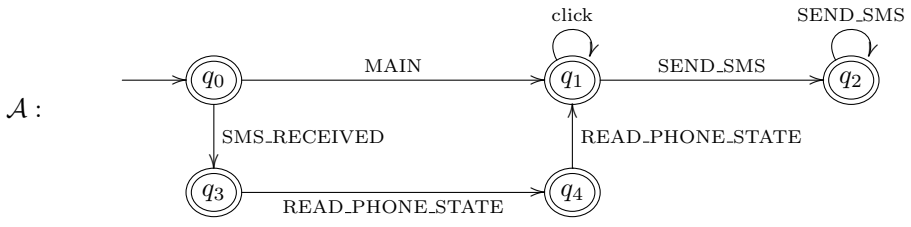
> *It sends SMS messages to a premium-rate number. It monitors received SMS messages and intercepts SMS messages. It may also steal information from the device.*

One of unwanted behaviours we have learned from malware instances in this family can be expressed as the regular expression: SMS_RECEIVED . SEND_SMS. The approach to learn these unwanted behaviours will be elaborated in Sections 4 and 5. It denotes the behaviour of sending an SMS message out *immediately* after an incoming SMS message is received. We generalise from this unwanted behaviour and construct the following automaton.



Here, we use the symbol Σ to denote the collection of events, actions, and permission-like phrases and the word "click" to denote that there is no interaction from the user. This automaton formalises the unwanted behaviour of sending an SMS message out after an incoming SMS message is received with no interaction from the user. It is actually a set of super-sequences of the sequence SMS_RECEIVED . SEND_SMS.

We now want to verify whether a target application has the above unwanted behaviour. Let us consider the following behaviour automaton $\mathcal{A}$. It is constructed from the byte-code of an Android application. Its source code and the method to construct behaviour automata will be given in Section 3.



It tells us: this application has two entries which are respectively specified by actions MAIN and SMS_RECEIVED; it will collect information like your phone state, then send SMS messages out; the behaviour of sending SMS messages can also be triggered by a user interaction, e.g., click a button, touch the screen, long-press a picture, etc., which is denoted by the word "click". All states in this automaton are accepting states since any prefix of an application's behaviours is one of its behaviours as well.

Because the intersection between $\mathcal{A}$ and ψ is not empty, we consider this application is unsafe with respect to the unwanted behaviour ψ. In Section 6, we will show that this verification against unwanted behaviours can improve the classification performance of new malware detecting.

3 Behaviour Automata

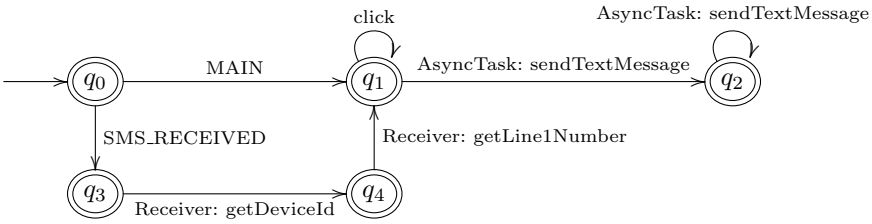
We use a simplified synthetic application to illustrate the construction of behaviour automata. This application will constantly send out the device ID and the phone number by SMS messages on background when an incoming SMS message is received. Its source code and part of its manifest file are as follows.

```
/* Main.java */
public class Main extends
  Activity implements View.OnClickListener {
  private static String info = "";
  protected void onCreate(Bundle savedInstanceState) {
    Intent intent = getIntent();
    info = intent.getStringExtra("DEVICE_ID");
    info += intent.getStringExtra("TEL_NUM");
    Task task = new Task();
    task.execute(); }
  public void onClick (View v) {
    Task task = new Task();
    task.execute(); }
  private class Task extends AsyncTask<Void, Void, Void> {
    protected Void doInBackground(Void... params) {
      while (true) {
        SmsManager sms = SmsManager.getDefault();
        sms.sendTextMessage("1234", null, info, null, null); }
      return null; }}}

/* Receiver.java */
public class Receiver extends BroadcastReceiver {
  public void onReceive(Context context, Intent intent) {
    Intent intent = new Intent();
    intent.setAction("com.main.intent");
    TelephonyManager tm = (TelephonyManager)
    getBaseContext().getSystemService(Context.TELEPHONY_SERVICE);
    intent.putExtra("DEVICE_ID", tm.getDeviceId());
    intent.putExtra("TEL_NUM", tm.getLine1Number());
    sendBroadcast(intent); }}

/* AndroidManifest.xml */
<activity android:name="com.example.Main" >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <action android:name="com.main.intent" />
  </intent-filter>
</activity>
<receiver android:name="com.example.Receiver" >
  <intent-filter>
```

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

As specified in AndroidManifest.xml, the Main activity can handle a specific Intent called "com.main.intent" and the Receiver will be triggered by an incoming SMS message (SMS_RECEIVED). After the Receiver collects the device ID and the phone number, it will send them out by a broadcast with the intent "com.main.intent". This broadcast is then handled by the Main activity in the method onCreate. Afterwards, SMS messages containing the device ID and the phone number are sent out on background in an AsyncTask.

We compile this application. From its byte-code we construct the following automaton.



This automaton is a collection of finite control-sequences of actions, events, and annotated API calls. Actions reflect what happens in the environment and what kind of service an application 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 application is doing anything we are interested in. For instance, getDeviceID, getLine1Number, and sendTextMessage are annotated API calls in the above example.

Notice that for a single behaviour there are often several related API methods. For example, getDeviceId, getLine1Number, and getSimSerialNumber are all related to the behaviour of reading phone state. We want to categorise API methods into a set of phrases, which describe behaviours of applications, so as to remove redundancy caused by API calls which indicate the same behaviour. This results in an abstract automaton, so-called a *behaviour automaton*. It has several advantages, including: more resilient to variants of behaviours, such as swapping two API calls related to the same behaviour; more compact than the original automata, which is good for human-understanding and further analysis, by reducing the number of labels on the edges. For instance, the behaviour automaton for the above example is the automaton $\mathcal{A}$ depicted in Section 2.

To construct such a behaviour automaton directly from an Android application, we have modelled 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, nested classes, interfaces, and runtime-registered listeners. We don't model registers, fields, assignments, operators, comparison, pointer-aliases, arrays or

exceptions. The choice of which features to model is a trade-off between efficiency and precision. 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 [12, 23, 25]. Compared with a simple list of API calls appearing in code, an automaton can capture more sophisticated behaviours. This is needed in practice, because: API calls appearing in code contain "noise" caused by dead code and libraries [7]; and, some unwanted behaviours only arise when some API methods are called in certain orders [17, 31, 43]. On the other hand, automata are less accurate than models which capture data-flows. However, it is much easier to generate behaviour automata using our tool for applications en masse than generating data-flows using tools like FlowDroid [10] or Amandroid [41]. In particular, people can annotate appealing API methods to generate compact behaviour automata more efficiently, rather than considering all data-dependence between statements.

In our implementation, we use an extension of permission-governed API methods generated by PScout [11] as annotations. The Android platform tools `aapt` and `dexdump` are respectively used to extract the manifest information and to decompile byte-code into assembly code, from which we construct the automaton. It took around two weeks to generate automata for 10,000 applications using a multi-core desktop computer.

4 Learning Unwanted Behaviours

Once a behaviour automaton has been constructed for each malware instance, we want to capture the common behaviour shared by malware, which is rarely seen in benign applications, so-called an *unwanted behaviour*. The space of candidate behaviours, which consists of the intersection and difference between behaviour automata, in theory, is exponential in the number of sample applications. To combat this, we approximate this space by searching for a "salient" subspace. The searching process is guided by the behavioural difference between malware and benign applications. We formalise this process as the following algorithm.

Function: construct_features (G, α)
Input: G – a group of behaviour automata
α – the lower bound on the classification accuracy
Output: salient sub-automata and their weights

1: $G_{i\in[0..N-1]} \leftarrow$ divide the set G into N groups
2: for $i \in [0..N-1]$
3: $F_i \leftarrow$ merge_features $(G_i, \emptyset)$
4: $s \leftarrow 2$
5: while $s \leq N$
6: for $i \in [0..N-1]$
7: $j \leftarrow i - (s/2)$
8: if $(i+1)\%s = 0$ then
9: $(F_i, \_), (F_j, \_) \leftarrow$ diff_features (F_i, α), diff_features (F_j, α)
10: $F_i \leftarrow$ merge_features (F_i, F_j)
11: elif $(i+1) > (N/s) \times s$ and $(i+1)\%(s/2) = 0$ then

```
12:        (F_i, _), (F_j, _) ← diff_features (F_i, α), diff_features (F_j, α)
13:    s ← s × 2
14:  return diff_features (F_{s/2−1}, α)
```

Function: merge_features (E, F)

```
1:  for e ∈ E
2:    for f ∈ F
3:      if f − e ≠ ∅ then F ← F ∪ {f − e}
4:      if f ∩ e ≠ ∅ then F ← F ∪ {f ∩ e}
5:      if f − e ≠ f and f ∩ e ≠ f then F ← F − {f}
6:      e ← e − f
7:    if e ≠ ∅ then F ← F ∪ {e}
8:  return F
```

Function: diff_features(F, α)

```
1:  D ← add an equal number of randomly-chosen benign applications
2:        into the set of malware instances from which F was collected
3:  W, acc ← train (D, F)
4:  if acc > α then F ← {f ∈ F | W_f ≠ 0}
5:  return F, W
```

The main process construct_features takes a collection G of behaviour automata as input and outputs a set F of salient sub-automata with their weights W. Here, a sub-automaton is *salient* if it is actually used in a linear classifier, i.e., its weight is not zero.

We randomly divide G into N groups: $G_0, \ldots, G_i, \ldots, G_{N-1}$. For each group, we construct sub-automata by computing the intersection and difference between automata within this group, i.e., merge_features $(G_i, \emptyset)$. This results in N feature sets $F_0, \ldots, F_i, \ldots, F_{N-1}$. The sub-automata in each set are disjoint. Then, we merge sub-automata from different groups, i.e., merge_features (G_i, G_j). This process stops until all groups have been merged into a single group.

Before merging sub-automata from two different groups, for each group, we train a linear classifier, i.e., train (D, F), using a training set D and a feature set F. This training set consists of behaviour automata of malware instances in the group and an equal number of behaviour automata of randomly-chosen benign applications. The input feature set F consists of disjoint sub-automata, which are constructed from behaviour automata of malware instances in the group. Then, if the classification accuracy acc on the training set is above a lower bound α, we return sub-automata with non-zero weights. Otherwise, we return all features in F. This process differentiates salient features by adding benign applications. It is formalised as the function diff_features.

In our implementation, we adopt L1-Regularized Logistic Regression [30, 39] as the training method. This is because this method is specially designed to use fewer features. We set the lower bound α on the classification accuracy to 90%. We have also designed and implemented a multi-process program to accelerate the construction, i.e., construct sub-automata for each group simultaneously. It took around one week to process $4,000$ malware instances using a multi-core desktop computer. At the end of the computation, we produced around $1,000$ salient sub-automata.

We will use these salient sub-automata to characterise unwanted behaviours. A straightforward way is to choose automata by their weights, for example, those with negative weights, i.e., $\{f \in F \mid W_f < 0\}$. More complex methods to capture unwanted behaviours will be discussed in next section.

5 Refining Unwanted Behaviours

To purify unwanted behaviours, we want to exploit the family names of malware instances to figure out the most informative ones, that is, to choose a small set of salient sub-automata to characterise unwanted behaviours for each family. Here are several candidate methods:

- *Top-n-negative*. For a linear classifier, intuitively, a feature with a negative weight more likely indicates an unwanted behaviour, and a feature with a positive weight more likely indicates a normal behaviour. This observation leads us to refine unwanted behaviours by using sub-automata with negative weights, i.e., choose the top-n features from the set $\{f \in F \mid W_f < 0\}$ by ranking the absolute values of their weights.
- *Subset-search*. For each malware family, we choose a subset X of salient sub-automata, such that it largely covers and is strongly associated with malware instances in this family. Formally, we use $Pr(f|X)$ to denote the probability of a malware instance belonging to a family f if all automata in X are sub-automata of the behaviour automaton of this instance, and $Pr(X|f)$ to denote the probability of all automata in X are sub-automata of the behaviour automaton of a malware instance if this instance belongs to f. We use F_1-measure as the evaluation function to look up subsets. i.e., $\frac{2Pr(f|X)Pr(X|f)}{Pr(f|X)+Pr(X|f)}$. Since exhaustively searching a power-set space is expensive, we adopt Beam Search [33, Chapter 6] to approximate the best K-subsets.
- *TF-IDF*. Another method is to consider features as terms, features from malware instances in a family as a document, and the multi-set of features as the corpus. We rank features by their TF-IDF (term frequency and inverse document frequency) and choose a maximum of m features to characterise unwanted behaviours of each family.

We collected more than $4,000$ malware instances from Malware Genome Project [1, 45] and Mobile-Sandbox [9, 37]. They have been manually investigated and organised into around 200 families by third-party researchers and malware analysts. We collected human-authorised descriptions for these families from their online analysis reports [1, 3, 5, 6, 32].

We then produce salient sub-automata from these malware instances by applying the algorithm in previous section and construct unwanted behaviours for each family by combining all methods discussed earlier. We list manual descriptions and learned unwanted behaviours of 10 prevalent families in Table 1.

A subjective comparison shows that these learned unwanted behaviours compare well to their manual descriptions. Also, they reveal trigger conditions of

manual description	*learned unwanted behaviours in regular expressions*
Arspam. Sends spam SMS messages to contacts on the compromised device [6].	1. BOOT_COMPLETED . SEND_SMS
Anserverbot. Downloads, installs, and executes payloads [1].	1. UMS_CONNECTED . LOAD_CLASS* . (ACCESS_NETWORK_STATE | READ_PHONE_STATE | INTERNET) . (ACCESS_NETWORK_STATE | READ_PHONE_STATE | INTERNET | LOAD_CLASS)*
Basebridge. Forwards confidential details (SMS, IMSI, IMEI) to a remote server [3]. *Downloads and installs payloads* [1, 6].	1. UMS_CONNECTED . (INTERNET | LOAD_CLASS | READ_PHONE_STATE | ACCESS_NETWORK_STATE)$^+$
Cosha. Monitors and sends certain information to a remote location [6].	1. MAIN . click . (click | ACCESS_FINE_LOCATION | DIAL)* . DIAL . (click | ACCESS_FINE_LOCATION | DIAL)* . (INTERNET | ϵ) 2. SMS_RECEIVED . (INTERNET | ACCESS_FINE_LOCATION)$^+$
Droiddream. Gains root access, gathers information (device ID, IMEI, IMSI) from an infected mobile phone and connects to several URLs in order to upload this data [1, 3].	1. PHONE_STATE . (ACCESS_NETWORK_STATE | READ_PHONE_STATE$^+$. INTERNET) . (ACCESS_NETWORK_STATE | INTERNET)*
Geinimi. Monitors and sends certain information to a remote location [6]. *Introduces botnet capabilities with clear indications that command and control (C&C) functionality could be a part of the Geinimi code base* [5].	1. ϵ | MAIN . click$^+$. VIBRATE . (click | VIBRATE)* . RESTART_PACKAGES . (MAIN . (click | VIBRATE)* . RESTART_PACKAGES)* 2. BOOT_COMPLETED . (ACCESS_NETWORK_STATE | click | INTERNET | RESTART_PACKAGES | ACCESS_FINE_LOCATION)$^+$
Ggtracker. Monitors received SMS messages and intercepts SMS messages [3]	1. MAIN . READ_PHONE_STATE 2. SMS_RECEIVED . SEND_SMS
Ginmaster. Sends received SMS messages to a remote server [32]. *Downloads and installs applications without user concern* [32].	1. BOOT_COMPLETED . LOAD_CLASS 2. MAIN . SEND_SMS
Spitmo. Filters SMS messages to steal banking confirmation codes [6].	1. NEW_OUTGOING_CALL . READ_PHONE_STATE . INTERNET . (INTERNET | ϵ)
Zitmo. Opens a backdoor that allows a remote attacker to steal information from SMS messages received on the compromised device [6].	1. SMS_RECEIVED . SEND_SMS 2. MAIN . READ_PHONE_STATE 3. MAIN . SEND_SMS

Table 1. Learned unwanted behaviours versus manual descriptions.

some behaviours, which were often lacking in manual descriptions. For example, the expression BOOT_COMPLETED.SEND_SMS denotes that after the device finishes booting, this application will send a message out; the expression UMS_CONNECTED.LOAD_CLASS means that when a USB mass storage is connected to the device, this application will load some code from a library or a hidden payload; and the unwanted behaviour for Droiddream shows that if the phone state changes (PHONE_STATE), this application will collect information then access Internet. Within manual descriptions displayed in Table 1, only two behaviours are not captured by learned unwanted behaviours: "gain root access" for Droiddream and the behaviour of Spitmo.

6 Evaluation: Detecting New Malware

We are concerned with whether unwanted behaviours can help improve the robustness of malware classification. As we will show in Table 2, a linear classifier using API calls and permissions as input features, which are popular input features for Android malware detectors [7, 9, 12, 15, 29, 44], performs badly on new malware instances (the testing set), although it has a very good classification performance on the validation set. In this section, we will show that unwanted behaviours improve the classification performance of new malware detection.

We collected 3,000 malware instances, which have been discovered before 2014, and 3,000 randomly-chosen benign applications. They include some famous benign applications, such as Google Talk, Amazon Kindle, and Youtube, and so on; and all malware instances from Malware Genome Project [1, 45] and most malware instances from Mobile-Sandbox [9, 37]. These malware instances have been manually investigated and organised into around 200 families by third-party researchers and malware analysts. By reading their online malware analysis reports [1, 3, 5, 6, 32], we learned what bad things would happen in these malware instances. We divided them into a training set and a validation set. Each of them consists of 1,500 malware instances across all families and 1,500 benign applications.

We test using a collection of 1,500 malware instances, which were released in 2014, and 1,500 randomly-chosen benign applications. These malware instances were from Intel Security and have been investigated by malware analysts. But, there is no family information or online analysis report about them. We have no idea of unwanted behaviours of these malware instances.

Permissions and lists of API calls appearing in code are extracted from these applications as input features to train classifiers as baselines.

We construct behaviour automata for all applications, then apply methods discussed in Sections 4 and 5 to learn unwanted behaviours from malware instances in the training set. Some behaviours of the application in question are not the same as unwanted behaviours, but, they often have unwanted behaviours as sub-sequences. For example, although the word SMS_RECEIVED.SEND_SMS is not accepted by the automaton $\mathcal{A}$ in Section 2, $\mathcal{A}$ accepts some sequences containing this word as a subsequence, i.e., SMS_RECEIVED.READ_PHONE_

STATE.READ_PHONE_STATE.SEND_SMS^{+}. To capture behaviours sharing the same patterns with unwanted behaviours, if a behaviour contains an unwanted behaviour as a sub-sequence, we consider this behaviour as unwanted as well. We call them *extended* unwanted behaviours. We check whether the intersection between the behaviour automaton of the application in question and an (extended) unwanted behaviour is not empty. We collect these verification results as input features to train the target classifiers.

For both baselines and target classifiers, we use L1-Regularized Logistic Regression [30, 39] as the training method.

feature	*validation (2011–13)*		*testing (2014)*		#salient/#feature
training (2011–13)	precision	recall	precision	recall	
signature-based features (baselines)					
permissions	89%	99%	53%	21%	59/175
apis	91%	98%	61%	15%	1443/52432
apis & permissions	93%	98%	**65**%	**15**%	735/52607
semantics-based features (targets)					
unwanted behaviours	66%	91%	53%	74%	634/886
ext. unwanted	75%	87%	69%	66%	581/886
ext. unwanted for families	72%	72%	**73**%	**66**%	**131/131**
mixed features					
all	**95**%	**99.5**%	65%	7.5%	870/61149

Table 2. Classification performance using different features.

The classification performance is reported in Table 2. It confirms that:

- Unwanted behaviours dramatically improve the classification performance on new malware instances. The classification performance using API calls and permissions as input features is very good on the validation set, i.e., the precision and recall are respectively 93% and 98%. However, this is just over-fitting to the training set, since its performance on the testing set is bad, in particular, the precision is 65% and recall is 15%. This means that a lot of new unwanted behaviours cannot be captured by API calls and permissions. By using the verification results against unwanted behaviours as input features, we improve the precision to 73% and the recall to 66%, as shown in the row of "ext. unwanted for families".
- Refining unwanted behaviours using the family names helps improve the classification performance of detecting new malware. The precision is increased from 69% (in the row of "ext. unwanted") to 73% (in the row of "ext. unwanted for families"), while maintaining the same recall. This refinement also helps reduce the number of features which are actually used in a linear classifier, in particular, totally 131 features were used, rather than 581 features.

7 Related Work

Our approach is close to the methodology proposed by Fredrikson et al. to synthesize malware specification [27]. In their work, a data dependence graph with logic constraints on nodes and edges was used to characterise an app's behaviours. From graphs of malware instances and benign apps they constructed so-called significant subgraphs that maximise the information gain. Then, several optimal collections of subgraphs were selected as specifications by using the formal concept analysis. They produced 19 specifications using 166 subgraphs constructed from 534 malware instances in 6 families and 39 benign apps. The evaluation was done on a collection consisting of 378 new malware instances and 28 benign apps. The main drawback of this method is its scalability. Also, the training and testing sets are very unbalanced, i.e., the number of benign apps is much less than that of malware instances.

Our approach is more scalable by using behaviour automata rather than data-flow models to approximate the behaviours of apps. Instead of using graph mining and formal concept analysis, we explore the weights assigned by the linear classifiers to accelerate the searching for salient sub-automata and refine the learned unwanted behaviours by exploiting the family names. These techniques enable our approach to deal with hundreds and thousands of malware instances spread in hundreds of families. Also, we tested on balanced datasets with equal numbers of malware instances and benign apps. By doing this the precision and recall are more comparable and convincing.

Machine learning methods have been applied to automatically detect Android malware [7, 9, 12, 15, 20, 28, 29, 38, 44]. DroidAPIMiner [7] uses refined API calls as features and relies on the KNN (k-nearest neighbours) algorithm. The method Drebin [9] extracts a broad range of features, such as permissions, components, API calls, and intents, from the manifest file and disassembled code, then trains a SVM classifier. The classifier produced by Yerima et al. [44] extracts permissions, API calls, and commands as features. Another interesting tool is CHABADA [29] which detects outliers (abnormal API usage) within clusters of applications by exploiting OC-SVM (one-class SVM). These clusters were grouped by descriptions of applications using LDA (Latent Dirichlet Allocation). Among others, the tool Dendroid [38] uses the cosine similarity between vectors of call graphs of malware to help group unknown malware samples into identified families. Similar ideas were applied in DroidLegacy [20] to detect piggybacking by exploiting the difference between API sets of modules in malware instances belonging to different families. Another interesting tool MAST [15] exploits MCA (Multiple Correspondence Analysis) to figure out indicative features. All of these tools and methods were trying to obtain good fits to a dataset by using different methods and variant kinds of features. The robustness of malware classification, in particular, the classifier specifically designed for new malware detection, has received much less consideration.

Aside from machine learning methods, static and dynamic analysis were applied to help malware detection in Android applications. Enck et al. [23] explored combinations of permissions to mitigate malware. Felt et al. [25] detected

over-privilege of permissions. Tools like TaintDroid [21], CopperDroid [34], and MonitorMe [31] are able to monitor dynamic behaviours of applications. The tool ComDroid [18] was designed for the detection of intent-based vulnerabilities. The tool Apposcopy extracts the weighted API dependency graphs to improve the classification performance [26]. More sophisticated tools like FlowDroid [10] and Amandroid [41] can identify paths from sources to sinks of sensitive information. All of these tools were designed to identify (or eliminate) malware (or unwanted behaviours) without learning or summarising useful properties for verification from their analysis results.

The idea of characterising applications' behaviours as automata is similar with the behaviour abstraction in [13, 42]. The behaviour automata are close to permission-event graphs [17], embedded call graphs [28], and behaviour graphs [43]. But, none of them has been exploited to automatically generate verifiable properties.

The unwanted behaviours can be considered as instances of security automata [35]. Our verification approach is the same as the automata-theoretic model checking [40]. More sophisticated approaches and finer formalisations can be found in the study of LTL model checking, e.g., Song et. al.'s work [36]. Totally 19 malicious properties for Android applications were manually constructed and specified as the first-order LTL formulae in [31]. Some benign and malicious properties specified in LTL were verified against hundreds of Android applications in [17]. But, none of these properties was automatically constructed.

Among others, Angluin's [8] and Biermann's [14] algorithms were developed to learn regular expressions from sample finite strings. To apply similar ideas in unwanted behaviour construction, we have to extract enough finite strings from applications. Comparing with the construction of behaviour automata, this will be more expensive.

8 Conclusion and Further Work

To learn compact, natural, and verifiable unwanted behaviours from Android malware instances is challenging and has not yet been considered. Compared with manually-composed properties, unwanted behaviours, which are automatically constructed from malware instances, will be much easier to be updated on the changes of behaviours exhibited in new malware instances. To the best of our knowledge, our approach is the first to automatically construct temporal properties from Android malware instances. We show that unwanted behaviours help improve the classification performance, in particular, they dramatically increase the precision and recall of detecting new malware. These unwanted behaviours can not only be used to eliminate potentially new instances of known malware families but also help people's understanding of unwanted behaviours exhibited in these families.

Some unwanted behaviours cannot be captured by our formalisation, e.g., gain root access, and some are not captured precisely enough, e.g., botnet controls. In further work, we want to extend the current formalisation to capture

more sophisticated behaviours precisely. We will also try to combine the output of dynamic analysis, e.g., traces produced by CopperDroid [34] or MonitorMe [31], with that of static analysis to approximate applications' behaviours. We will explore whether properties expressed in LTL are needed in the practice of malware detection and whether it is possible to learn them from malware instances. The verification method adopted in this paper is straightforward and simple. More efficient and complex methods, e.g., the method discussed in [36] and model checking pushdown systems [24], will be considered in future. Except unwanted behaviours, we will investigate whether other methods can help improve the robustness of malware classifiers, e.g., semi-supervised learning [16]. On the other hand, since unwanted behaviours are context-sensitive, i.e., an unwanted behaviour in a group of applications might be normal in another, we want to organise applications into groups sharing similar behaviours and construct unwanted behaviours for each group. Also, in practice, many malware instances have no family information. We will explore methods to organise malware instances when the family information is unavailable, e.g., clustering by compression [19]. It is also interesting to study whether unwanted behaviours can convince people of the automatic malware detection.

References

1. Malware Genome Project. http://www.malgenomeproject.org/, 2012.
2. Mobile Adware and Malware Analysis. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/madware_and_malware_analysis.pdf, 2013.
3. Forensic Blog. http://forensics.spreitzenbarth.de/android-malware/, 2014.
4. McAfee Mobile Security Report. http://www.mcafee.com/us/resources/reports/rp-mobile-security-consumer-trends.pdf, 2014.
5. Juniper Networks. https://www.juniper.net/security/auto/includes/mobile_signature_descriptions.html, 2015.
6. Symantec security response. http://www.symantec.com/security_response/, 2015.
7. Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *SecureComm*, pages 86–103. Springer, 2013.
8. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.
9. D. Arp et al. Drebin: Efficient and explainable detection of Android malware in your pocket. *NDSS*, pages 23–26, 2014.
10. 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.
11. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *CCS*, pages 217–228, 2012.
12. D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *CCS*, pages 73–84, 2010.
13. P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *Runtime Verification*, LNCS 6418, pages 168–182. Springer, 2010.

14. A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.
15. S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: Triage for market-scale mobile malware analysis. In *WiSec*, pages 13–24, 2013.
16. O. Chapelle, B. Schlkopf, and A. Zien. *Semi-Supervised Learning*. The MIT Press, 1st edition, 2010.
17. K. Z. Chen et al. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS*, 2013.
18. E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.
19. R. Cilibrasi and P. M. B. Vitnyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51:1523–1545, 2005.
20. L. Deshotels, V. Notani, and A. Lakhotia. DroidLegacy: Automated familial classification of Android malware. In *PPREW*, 2014.
21. W. Enck et al. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, 2014.
22. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX Security Symposium*, 2011.
23. W. Enck, M. Ongtang, and P. D. McDaniel. On lightweight mobile phone application certification. In *CCS*, pages 235–245, 2009.
24. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, LNCS 1855, pages 232–247. Springer, 2000.
25. A. P. Felt et al. Android permissions demystified. In *CCS*, pages 627–638, 2011.
26. Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *SIGSOFT FSE*, 2014.
27. M. Fredrikson et al. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 45–60, 2010.
28. H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *AISec*, pages 45–54, 2013.
29. A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *ICSE*, 2014.
30. T. Haštie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2001.
31. J.-C. Kuester and A. Bauer. Monitoring real android malware. In *Runtime Verification 2015*, Vienna, Austria, sep 2015.
32. McAfee Threat Center. http://www.mcafee.com/uk/threat-center.aspx, 2015.
33. P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers Inc., 1st edition, 1992.
34. 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 (EUROSEC)*, 2013.
35. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.
36. F. Song and T. Touili. LTL model-checking for malware detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 416–431. 2013.
37. 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.

38. G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, 41(4, Part 1):1104 – 1117, 2014.
39. R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.
40. M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32(2):183–221, 1986.
41. 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.
42. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, July 2002.
43. C. Yang et al. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *ESORICS*, September 2014.
44. 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.
45. Y. Zhou and X. Jiang. Dissecting Android malware: characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109, 2012.