6 MACHINE LEARNING FEATURES

We've explored the two kinds of analysis required to understand an Android app: static and dynamic. We've also seen how a human security analyst can go back and forth between static and dynamic analysis to pinpoint the locations at which dangerous behavior occurs.

However, machine learning algorithms can't perform the "back and forth" behavior of a human analyst. Because they can't choose to explore one part of the code more than another part, they must associate a feature vector with each app, regardless of whether it is malicious or benign, and then use a previously trained model to make a prediction about it. This means they must determine in advance what to include in the feature vector.

In this chapter, we first describe how to turn static and dynamic sources of information into input for machine learning algorithms, enabling us to scale our malware detection efforts to millions of APKs. Then we explore four novel types of features that are harder for attackers to evade or reverse engineer, yet robust enough to detect malware with high accuracy. These detection techniques take into account the fact that malware developers often understand the static and dynamic analysis methods used by security experts and can apply this knowledge to evade detection.

Static Features

The first class of features we can associate with Android apps is based on a static analysis of the code. Unlike with data gleaned through human-based static analysis, software can easily extract their values. These features are immutable, in the sense that once we train a predictive model with a given set of features, we must stick with them when using the predictive model (however, new features can be added and old features removed when retraining).

There are several files and folders inside an APK whose properties and content we can turn into machine learning features. One source of features is the *AndroidManifest.xml* file that every APK contains in its root directory. As discussed in Chapter 3, the manifest file defines the structure and metadata of the Android application, including the package name and app version. It might also include XML nodes that describe the app's basic behavior, as well as the permissions requested by the application. Listing 6-1 shows a snippet from the manifest file of a malware app, Fakebank *com.a* (v152, 0add), that references XML nodes.

```
<receiver android:label="@string/app2"
android:name="com.p004a.p005a.DeAdminReciver"
android:permission="android.permission.BIND_DEVICE_ADMIN"
android:description="@string/app2">
<meta-data android:name="android.app.device_admin"
android:resource="@xml/an"/>
```

Listing 6-1: XML nodes in the Android manifest associated with the Fakebank app

We might also find features in the Java source code folder. In Java apps, this folder is part of the original code and is not present in the compiled APK file. Other folders of interest are *Res*, *lib*, and *assets*. The *Res* folder contains all non-code resources used by the application, such as XML layouts and images. The *lib* folder is tricky, as its purpose changes after compilation: in Android source code, it's often used to store common files, utility classes, and imported dependencies associated with applications, while in compiled APK files it stores native code files used by the application. The *assets* folder might include a wide range of files, such as text, XML, fonts, music, and video. Another source of features is the *build.gradle* file, which includes build-related configurations. It is present only during development and not included in the final APK.

We can define two versions of many features. A *binary* version of a feature is set to 0 or 1 depending upon whether the feature does or doesn't occur. For instance, we might associate a feature with an API function call. If an app makes at least one call to that API in its code, we'd set the binary version of the feature to 1; otherwise, we'd set it to 0. A *statistical* version of the same feature, on the other hand, might reflect the number of calls the app makes to the API. Alternatively, it might record the number of times the API function is invoked with certain inputs and return a statistical quantity, such as the mean, median, or standard deviation of the results. The following features commonly appear in the literature on machine learning–based malware detection:

Permissions

We can design a number of features related to the permissions that an application requests. We might, for example, create a binary feature for each permission. We could also define statistical features corresponding to the number of normal, signature, and dangerous permissions requested. According to the official Android developer site, dangerous permissions are those that either involve the private data of users or could possibly affect such private user data. For instance, we've already seen that the *com.bp.statis.bloodsugar* malware discussed in Chapter 3 requests the `READ_CONTACTS` permission even though there is little reason to believe that a blood sugar monitoring app needs access to a user's contacts.

Activities

As discussed in Chapter 3, activities implement the visual interface of an Android app and are declared in the manifest file. We could create a set of binary features to indicate whether each activity is used or not. The total number of activities is also a potential feature.

Services

Apps use services to implement long-running background operations that facilitate interactions with the system. As in the case of activities, we can define binary features indicating whether each service is used or not. Moreover, we can define simple counts and statistical features. For instance, in the case of the *com.bp.statis.bloodsugar.PE* service discussed in Chapter 3, we might set this value to 1, as there is little reason for a blood sugar app to listen to incoming notifications from all of the apps in the system.

Content providers

A content provider encapsulates data and gives it to other applications. For each content provider, we might have a feature set to 0 if it doesn't exist and 1 if it does exist in the app. We can also create a feature for the number of content providers the app uses.

Broadcast receivers

The broadcast receiver component of an application enables it to receive broadcast messages from the system or other applications. As in the preceding cases, we can create binary features, counts, and statistical features for these receivers. However, while it is easy to find broadcast receivers declared in the manifest file, it is not always easy to find those declared at runtime, especially as they may be part of encrypted or obfuscated code. Moreover, some apps might want to register a `RECEIVE_SMS` receiver, which enables them to intercept incoming SMS traffic (for example, one-time passwords or alerts of suspicious activity).

Intent filters
Activities, services, and broadcast receivers can use intent filters to specify the kinds of operations to which they will respond. In the case of broadcast receivers, intents specify the types of broadcasts that they can handle. As in the preceding cases, we can define and extract binary features and statistical features for intents.

API calls
The Android platform provides a set of API packages that developers can use to build applications. We can create binary features for each API package (based on whether it is called or not) as well as for each class within those packages (based on whether the class is called or not). In addition, numeric features for an API package might track the number of times the app calls a class within a package or a function within a class. We'll provide a detailed introduction to API features later in this chapter, as we can use them to generate more advanced features.

Network elements
An Android application's source code may contain numerous network elements, such as IP addresses, URLs, and hostnames. We can collect these elements to generate binary features and statistical features (for example, the number of hostnames listed in the file). We may also want to use the number of external URLs referenced in the code as a feature.

The malware author might try to make static analysis difficult through a variety of instruments. These could include using unintelligible names for variables, encrypting parts of the code, and using other obfuscation methods such as reflection (see Chapter 3). We can also define static features to describe whether such phenomena exist in the app code, as well as their frequency of occurrence.

Dynamic Features

We can turn the results of our dynamic analysis into machine learning features, too. As covered in Chapter 4, dynamic analysis focuses on observed runtime properties and behavior of applications. Consequently, the features derived from it describe events that were actually observed rather than the more speculative features derived from the static analysis of code.

To generate many of these features, we must feed some set of inputs to the app, such as interactions that the app has with the user (an example is the `monkey` command discussed in Chapter 4). We might run the app using the first input and generate some results, then run it with the second input and generate more results, continuing the process until we've exhausted all inputs in the set. We can also extract features by analyzing the network traffic generated when the app is run through programs such as tcpdump and Wireshark.

The following dynamic features for the Android platform have been widely discussed in the literature:

Services

We can generate dynamic features to record each started service. These may be binary (based on whether the app starts that service or not) or numeric (for example, the average number of calls to the service across the set of inputs). The total number of services started can also be a feature. Additionally, we could associate a sequence of services with a feature by recording whether the app ever invokes that sequence (a binary feature) or how many times an app invokes it on average across the set of inputs (a numeric feature).

The `DexClassLoader`

This is a standard Android API used to load classes from *.jar* and *.apk* files that contain a *classes.dex* file. Malicious apps frequently use this API to evade static analysis because it lets them execute code that didn't come from the application's source code (one example is the Xenomorph malware family discussed in Chapter 4). We could create a feature that is set to 1 if the app calls `DexClassLoader` and to 0 otherwise. Additional features could be defined based on the count of calls, *n*-gram sequences, and other statistics.

Permissions

We can create binary features that record whether the app invokes an API that requires some permission, even if the permission doesn't appear in the application code itself. Although Android apps must explicitly declare any permissions that they request within the manifest file, they might try to circumvent this requirement by acquiring permissions in different ways. One strategy is to use a covert channel, such as the communication between multiple APKs, to share information. This behavior poses a challenge to dynamic analysts, as their lab setups must be able to run multiple interacting apps at the same time. As in the previous cases, we can also generate statistical features and *n*-gram features based on permissions. For instance, in the Xenomorph malware, we would record the fact that it invoked the accessibility API by setting that value to 1.

Data leaks

An app might sometimes leak a user's personal data, be it accidentally, because the app is poorly coded, or intentionally, in an attempt to steal the data. We can generate features that reflect the leaked content.

Use of cryptography

We can define a feature that tracks whether an app performs any cryptographic operations. When an APK executes encryption operations (for example, to store encrypted files), the sandbox used to run it can track and record this. We might set a binary feature to 1 if the app generates any encrypted files during execution and set it to 0 otherwise. We see this behavior in the Xenomorph app; see the `encryptMessage` function in Listing 4-5, which the app could invoke zero or more times during its execution in a sandbox environment.

Network activities

We can use a set of features to keep track of operations that open or close network sockets by recording the destination host. We might also create features based on the data received from the network, as well as the source of the data and any data that the application sends to others on the network.

Sending SMS messages

When an app sends text messages during its execution phase, we can record the identity of the recipient and the message's content to use as features. We can also count the total number of messages sent during the execution or define a binary feature that we set to 1 if the app sends any messages at all.

Phone calls

Malicious Android apps sometimes make phone calls (for example, to premium rate numbers). In such cases, we can define features to store the numbers called or use a binary feature to record the fact that some external numbers were called.

Answered intents

We can capture the intents to which the application responds during its execution and record these as dynamic features.

Files

We might create features to record the names of any library files that the app uses. Also, when the application reads or writes to specific files, we can capture the filename and the content, then generate features based on these.

Method Call Features (A Weak Tactic)

To go beyond basic static and dynamic data, some researchers have turned to API method calls as potential features. The Android platform provides a set of API packages that developers can use to access a host of valuable functionality. For example, the *android.accessibilityservice* package can help users with disabilities interact with Android devices. However, malware developers can also use it, and they widely abuse it.

Each API package contains a number of classes, and each class has its own methods that we can use to define new features for our models. To create features using the 171 API packages in Android API 23, for example, we might build a 171-dimensional feature vector for each Android app to capture the frequency with which that app calls the methods from each package. For instance, if some API package includes 40 methods belonging to different classes and an app calls each of them twice, the corresponding feature value would be 40×2, or 80.

These API feature values can vary greatly. For instance, consider the 171 API feature values associated with a goodware sample called *ESPN 6.0.4*. The largest of these feature values is 161,698, while the smallest is 0, producing a standard deviation of 10,488.26. By contrast, another goodware sample, *com.hancom.office.editor* (v1, 75d1), has 6 as its largest API feature value and 0 as its smallest, with a standard deviation of only 0.61. You might have the instinct to normalize feature values to account for this difference, but normalizing isn't necessary because good machine learning algorithms will automatically determine which values of a given feature help create good separators between malware and goodware.

While you'll find these API-based features used in the literature, malware developers can evade them easily. Zhengcuan Cai and Roland Yap studied 57 Android antivirus tools in their 2016 paper "Inferring the Detection Logic and Evaluating the Effectiveness of Android Anti-Virus Apps." They found that malicious hackers can easily uncover the detection logic in antivirus apps that use static analysis alone, enabling them to evade detection. For instance, in this case the developers could include a bunch of dummy calls to API features in order to change their app's 171-dimensional feature vector. Likewise, obfuscation methods such as reflection and dynamic code loading can lower an app's feature counts. The feature counts of particularly well-hidden method calls might even drop to zero if static analysis doesn't find those calls (which would be the case if, for instance, the calls were in an encrypted section of code). For completeness, machine learning models should include both static and dynamic features of API calls.

By contrast, advanced features, based on techniques like triadic suspicion graphs, landmarks, feature clustering, and correlation graphs, are highly effective in identifying malicious Android apps. Experiments have shown that such features are harder for malicious hackers to evade, in part because it is hard for them to determine exactly how these features are used in a detection system. The remainder of this chapter introduces these advanced features.

Triadic Suspicion Graph Features

Rather than using API method calls on their own, we can generate a more robust group of features derived from a special class of graphs called *triadic suspicion graphs (TSGs).* Essentially, a TSG aims to understand the differences between the use of an API package by goodware on the one hand and different types of malicious apps on the other hand. Figure 6-1 is an illustration of a TSG that compares goodware to banking trojans. We'll walk through its elements in the paragraphs that follow.

A TSG is made up of vertices connected by edges. In this context, the TSG contains three kinds of vertices: the complete set of API package calls defined in the Android API, the sampled goodware, and the sampled malware, randomly drawn from some larger collections of goodware and malware, respectively. The TSG's edges are defined as follows:

1. For each goodware g and each API package call a, if g calls a method from a at least once, there is an edge from g to a.
2. For each pair of API package calls a_1 and a_2, if a_1 calls any method from a_2, there is an edge from a_1 to a_2.

3. For each malware b and each API package call a, if b calls any method from a at least once, there is an edge from b to a.

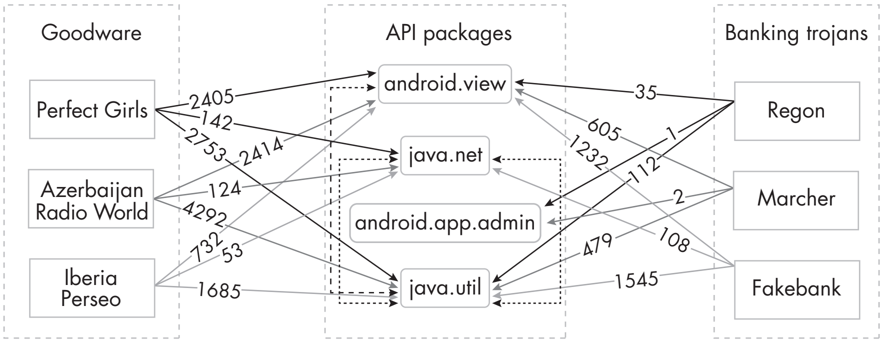


Figure 6-1: A partial TSG containing three goodware samples

The goodware and malware collections don't need to be fixed. An analyst might use one sampling in one week, switch to another in the next week, and keep doing so regularly in order to present a moving target. Varying the sets changes the attack surface and makes it harder for an adversary to guess the precise nature of the defense.

We also suggest keeping the size of the sets relatively small, and varying it as well. For example, if we had access to 1 million goodware samples and 10,000 malware samples, we might select only, say, 1,000 samples for each of the groups in the first week, 1,322 in the next week, 1,127 in the third week, and so forth. Frequently modifying the sample sizes is another way to keep the attacker in the dark about the nature of the defenses being used; however, the number of samples in the two sets should be approximately the same.

Once we've determined the vertices and edges in a TSG, we weight the edges using a weight function. In this context, the weights reflect the number of times an app calls a corresponding API package's methods. For any edge from a goodware or malware app v to an API package a, we use $f(v, a)$ to denote the number of times v calls methods from a. The following equations demonstrate five plausible definitions of a weight function w. Functions w_1, w_2, and w_3, respectively, represent linear, quadratic, and cubic relationships between the API package call frequency and the edge weight, while w_4 and w_5 capture other possible nonlinear relationships:

$$w_1(v, a) = f(v, a)$$
$$w_2(v, a) = f(v, a)^2$$
$$w_3(v, a) = f(v, a)^3$$
$$w_4(v, a) = \sqrt{f(v, a)}$$
$$w_5(v, a) = \ln\left(f(v, a) + 1\right)$$

Having different definitions is useful because most machine learning algorithms are very sensitive to the input features and can't always correctly infer the most accurate nonlinear relationships between data points using the modeling framework alone.

We set the weights of edges between pairs of API package calls to the same default value, 1. This is because we are more interested in whether a specific edge exists than in the frequency with which one API package calls another within the Android API, as attackers can't control these relationships.

You can see the weighted edges in Figure 6-1, which uses the function w_1, as well as directional arrows to show the calling relationships among pairs of API packages. Now you can observe that none of the three goodware samples call the API package `android.app.admin`, while two of the three banking trojans call it a few times. These sorts of patterns might help us identify malicious apps.

Suspicion Scores

With the TSG defined, we can now calculate the *suspicion score* of an API package. In short, we rank an API package that is frequently invoked by malware but not by goodware as more suspicious than one that is frequently invoked by goodware but not by malware. Suspicion scores alone aren't enough to predict whether an Android app is malicious or not, but they do generate a set of features that might be able to provide good predictive performance. Moreover, as the malware developer won't know the reference sets and weight functions used to create the TSGs, they can't easily evade detection frameworks that use them.

We define 12 possible suspicion scoring functions, sus_1 through sus_{12}. Having multiple candidate functions ensures that we are less prone to overfitting a predefined model. When we supply these scores and other features as input, machine learning techniques can tell us which suspicion scoring function is best able to differentiate benign apps from malicious ones. You might notice that the function definitions, shown next, are closely related to the weight functions w_1 through w_5:

$$sus_1(a_j) = \frac{\frac{\sum_{i=1}^{n} I(b_i, a_j)}{n}}{\frac{\sum_{i=1}^{n} I(b_i, a_j)}{n} + \frac{\sum_{i=1}^{m} I(g_i, a_j)}{m}}$$

$$sus_2(a_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i, a_j)}{n}}{\frac{\sum_{i=1}^{n} f(b_i, a_j)}{n} + \frac{\sum_{i=1}^{m} f(g_i, a_j)}{m}}$$

$$sus_3(a_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i, a_j)^2}{n}}{\frac{\sum_{i=1}^{n} f(b_i, a_j)^2}{n} + \frac{\sum_{i=1}^{m} f(g_i, a_j)^2}{m}}$$

$$sus_4(a_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i, a_j)^3}{n}}{\frac{\sum_{i=1}^{n} f(b_i, a_j)^3}{n} + \frac{\sum_{i=1}^{m} f(g_i, a_j)^3}{m}}$$

$$sus_5(a_j) = \frac{\frac{\sum_{i=1}^{n} \sqrt{f(b_i, a_j)}}{n}}{\frac{\sum_{i=1}^{n} \sqrt{f(b_i, a_j)}}{n} + \frac{\sum_{i=1}^{m} \sqrt{f(g_i, a_j)}}{m}}$$

$$sus_6(a_j) = \frac{\frac{\sum_{i=1}^{n} \ln\left(f(b_i, a_j) + 1\right)}{n}}{\frac{\sum_{i=1}^{n} \ln\left(f(b_i, a_j) + 1\right)}{n} + \frac{\sum_{i=1}^{m} \ln\left(f(g_i, a_j) + 1\right)}{m}}$$

$$sus_7(a_j) = \frac{\frac{\sum_{i=1}^{n} I(b_i, a_j)}{\sum_{a_j} \sum_{i=1}^{n} I(b_i, a_j)}}{\frac{\sum_{i=1}^{n} I(b_i, a_j)}{\sum_{a_j} \sum_{i=1}^{n} I(b_i, a_j)} + \frac{\sum_{i=1}^{m} I(g_i, a_j)}{\sum_{a_j} \sum_{i=1}^{m} I(g_i, a_j)}}$$

$$sus_8(a_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i, a_j)}{\sum_{a_j} \sum_{i=1}^{n} f(b_i, a_j)}}{\frac{\sum_{i=1}^{n} f(b_i, a_j)}{\sum_{a_j} \sum_{i=1}^{n} f(b_i, a_j)} + \frac{\sum_{i=1}^{m} f(g_i, a_j)}{\sum_{a_j} \sum_{i=1}^{m} f(g_i, a_j)}}$$

$$sus_9(a_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i, a_j)^2}{\sum_{a_j} \sum_{i=1}^{n} f(b_i, a_j)^2}}{\frac{\sum_{i=1}^{n} f(b_i, a_j)^2}{\sum_{a_j} \sum_{i=1}^{n} f(b_i, a_j)^2} + \frac{\sum_{i=1}^{m} f(g_i, a_j)^2}{\sum_{a_j} \sum_{i=1}^{m} f(g_i, a_j)^2}}$$

$$sus_{10}(a_j) = \frac{\frac{\sum_{i=1}^{n} f(b_i, a_j)^3}{\sum_{a_j} \sum_{i=1}^{n} f(b_i, a_j)^3}}{\frac{\sum_{i=1}^{n} f(b_i, a_j)^3}{\sum_{a_j} \sum_{i=1}^{n} f(b_i, a_j)^3} + \frac{\sum_{i=1}^{m} f(g_i, a_j)^3}{\sum_{a_j} \sum_{i=1}^{m} f(g_i, a_j)^3}}$$

$$sus_{11}(a_j) = \frac{\frac{\sum_{i=1}^{n} \sqrt{f(b_i, a_j)}}{\sum_{a_j} \sum_{i=1}^{n} \sqrt{f(b_i, a_j)}}}{\frac{\sum_{i=1}^{n} \sqrt{f(b_i, a_j)}}{\sum_{a_j} \sum_{i=1}^{n} \sqrt{f(b_i, a_j)}} + \frac{\sum_{i=1}^{m} \sqrt{f(g_i, a_j)}}{\sum_{a_j} \sum_{i=1}^{m} \sqrt{f(g_i, a_j)}}}$$

$$sus_{12}(a_j) = \frac{\frac{\sum_{i=1}^{n} \ln(f(b_i, a_j) + 1)}{\sum_{a_j} \sum_{i=1}^{n} \ln(f(b_i, a_j) + 1)}}{\frac{\sum_{i=1}^{n} \ln(f(b_i, a_j) + 1)}{\sum_{a_j} \sum_{i=1}^{n} \ln(f(b_i, a_j) + 1)} + \frac{\sum_{i=1}^{m} \ln(f(g_i, a_j) + 1)}{\sum_{a_j} \sum_{i=1}^{m} \ln(f(g_i, a_j) + 1)}}$$

These suspicion score functions all make use of an indicator function $I(v_1, v_2)$ to denote the existence of an edge from vertex v_1 to v_2, where $v_1, v_2 \in \mathcal{A} \cup \mathcal{G} \cup \mathcal{M}$. In other words, if it is the case that $f(v_1, v_2)$ is greater than 0, then $I(v_1, v_2)$ equals 1; otherwise, it is 0. (In fact, we could treat the $I(v, a)$ function for edges from apps to API packages as another kind of weight function.) We use n to denote the number of malware samples and m to denote the number of goodware samples.

For example, according to the first function, sus_1, if the API package a_j is called by 100 malicious apps b and 10 goodware apps g from our samples, we reasonably consider apps that invoke this API package to be more suspicious than ones that do not. The definition in sus_7 is another way of capturing the same intuition: that an API function that is more extensively called by malicious apps than by benign ones will have a higher suspicion score. Equations sus_7 through sus_{12} make similar assumptions to sus_1 through sus_6 except that they evaluate the suspicion score of one API package with respect to all API packages rather than by itself.

The Suspicion Rank

Suspicion scores label a single Android API package call by looking at how malware and goodware each call that package. However, a package might itself make calls to other packages within the Android API. If a package $P1$ makes lots of calls to another package Q that has a high suspicion score, we should rank the first package as more suspicious than a package $P2$ that makes no calls to packages that have a high suspicion score.

The situation is a bit like an individual making lots of calls to a drug dealer. Even if the individual isn't deemed suspicious in their own right, the fact that they're in regular contact with a drug dealer makes them so. This is precisely the intuition behind Google Search's famous PageRank algorithm, which captures the importance of a web page by considering the importance of the web pages that link to it.

In fact, we can combine our suspicion scores with PageRank to define a family of suspicion ranking functions that capture these intuitions. PageRank calculates the importance of web pages using the following formula:

$$PR(v) = \frac{1-d}{N} + d \times \sum_{(u,v)\in E} PR(u)out(u)$$

Here, E is the set of edges in the web; N is the total number of nodes, or vertices, in the web; $d \in [0, 1]$, called the damping factor, is usually set to 0.85; and $out(u)$ is the *out-degree* of node u, or the number of edges that leave it. The $\frac{1-d}{N}$ expression captures the probability that a user will reach web page v by explicitly entering its address into a browser, while the remaining part of the expression is intended to capture the probability of a user reaching page v by following links.

In the following, we define the suspicion rank for an Android API package a with respect to a fixed suspicion scoring function, *sus*. We could use any of the functions described earlier in this chapter, or an entirely new one, as long as it associates a suspicion score with each function in the Android API:

$$SR_{sus}(a) = \frac{1-\delta}{|\mathcal{A}|} + \delta \times \sum_{a'\in\mathcal{A},(a',a)\in E} \frac{sus(a') \times SR_{sus}(a')}{out(a')}$$

The parameter $\delta \in [0, 1]$ is a damping factor similar to PageRank's d. In practice, we set it to 0.85, as is usually done with PageRank. The value a' is any package invoked by the package a, and the $out(a')$ value is the out-degree of the node in the TSG corresponding to a'. In other words, it represents the number of API packages invoked by a'.

Readers might have noticed that the definition of the suspicion rank mainly relies on a small portion of the TSG—namely, the vertices representing API packages and the edges between them. As a result, this structure is independent of the choice of apps in the goodware and malware sets, and adversaries can't manipulate it, because it (that is, the Android API) is publicly disclosed in the Android code and documentation. This approach differs from the function call graphs described in previous works, which usually depend on the sequence of operations within specific individual apps and so lack randomness, a key element in keeping malware developers guessing. We list some of these alternative approaches in "Further Reading" on page 202.

TSG Features

The preceding two sections define ways to calculate suspicion scores and suspicion ranks for API packages in a given TSG. In total, we have 24 kinds of suspicion-based scores associated with each API package. Researchers can add new ones if they wish. Next, we must use these suspicion-based scores to generate what we call *TSG features* for Android apps. These features capture the package call behavior of all apps, meaning an app doesn't have to be in either the malware or the goodware sample set to have TSG features.

To generate these features, we first rank the API packages in descending order according to their suspicion score and suspicion rank results. Theoretically, the higher the rank of an API package, the more suspicious it is. However, we will have noise, perhaps stemming from the choice of sample applications. Therefore, instead of directly using the ranked package list, we apply a window-based segmentation to it before deriving TSG features.

The basic idea of *window-based* segmentation is to use an integer W that is greater than 1 to segment the list into a number of buckets, starting from the beginning of the list. As shown in Figure 6-2, each bucket (except possibly the last one) contains W API packages with similar suspicion-based scores or ranks.

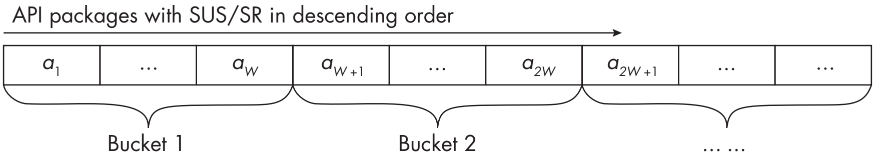


Figure 6-2: A window-based API package ranking by descending suspicion scores and ranks

Suppose API packages a_1 through a_W are in the same bucket, and suppose that the corresponding API feature values of an app are f_1 through f_W. For each bucket, we can calculate a TSG feature via one of the following six methods:

Binary value Does this app call any API packages in this bucket? If so, this binary feature is 1; otherwise, it is 0.

Number of API packages How many API packages in this bucket does the app call? The feature value is an integer $\sum_{j=1}^{W} I(f_j)$, where the function $I(f_j) = 1$ is $f_j > 0$; otherwise, it equals 0.

Maximum frequency value Of the call frequencies from the app to all API packages in the bucket, what is the maximum value? The feature value is an integer $\max_{j=1}^{W} f_j$.

Median frequency value Among the call frequencies, what is the median value? The feature value is an integer $\text{median}_{j=1}^{W} f_j$.

Sum of frequencies How many times in total does this app call API packages in this bucket? The feature value is an integer $\sum_{j=1}^{W} f_j$.

Weighted sum Based on the frequency sum, what would the value be if we took the suspicion score given by function ρ as the corresponding weight? This feature is a real value $\sum_{j=1}^{W} \rho_j f_j$, where ρ_j stands for the suspicion score of API package a_j.

To illustrate how these features work, consider the small dataset with three banking trojans and three goodware samples we showed earlier in this chapter. Suppose Table 6-1 shows the frequency with which the malware sample Regon calls the four API packages.

Table 6-1: Frequency of API Package Calls by Regon

	android.view	***java.net***	***android.app.admin***	***java.util***
Frequency	35	0	1	112

If we use sus_1 as our suspicion scoring function, we could sort the packages based on their suspicion scores, in descending order, as shown in Table 6-2.

Table 6-2: Suspicion Scores of the Packages Called by Regon

	android.app.admin	***android.view***	***java.util***	***java.net***
Suspicion score	1	0.5	0.5	0.25

Suppose we now use $W = 2$ as the window size. In this case, there are two buckets, the first containing *android.app.admin* and *android.view* and the second containing *java.util* and *java.net*. We derive the following feature values for Regon from the first bucket: a binary value of 1, an API package number of 2, a sum of frequencies of 36, a maximum frequency value of 35, a median frequency value of 18, and a weighted sum of $1 \times 1 + 0.5 \times 35$, or 18.5. The values of the features generated by the second bucket are 1, 1, 112, 112, 56, and 56, respectively.

Now suppose we repeat this process using both the suspicion scoring function sus_1 and the suspicion ranking formula. Table 6-3 shows the resulting suspicion ranks after sorting.

Table 6-3: Suspicion Ranks for the Packages Called by Regon

	java.util	***java.net***	***android.view***	***android.app.admin***
Suspicion rank	0.1025	0.0811	0.0375	0.0375

These suspicion ranks generate the following feature values for Regon from the first bucket: a binary value of 1, an API package number of 1, a sum of frequencies of 112, a maximum frequency of 112, a median frequency of 56, and a weighted sum of $0.1025 \times 112 + 0.0811 \times 0$, or 11.48. For the API calls in the second bucket, Regon has corresponding feature values 1, 2, 36, 35, 18, and 1.35.

In this example, we generated TSG features for Regon based on a subset of Android API packages and a part of the complete TSG. In a real implementation, however, we might use all 171 API packages, 24 different suspicion scoring functions, and 6 methods for computing TSG features for each function. As a result, if we use a *W* of 10, we could generate 2,592 TSG features for each app.

In addition, because we control the *W* parameter, we can vary it in several ways. For instance, if we have four API packages with the suspicion scores 0.9, 0.3, 0.29, and 0.2, we could divide them into two evenly sized buckets, (0.9, 0.3) and (0.29, 0.2). Alternatively, we could group similar scores together by using a variable window size to segment them into two buckets, (0.9) and (0.3, 0.29, 0.2). Using window size in this way has an advantage: it introduces yet another complication for the adversary. If an attacker changed the number of calls made in a piece of malware to classes in one or two Android API packages, it wouldn't have a huge impact on how features were derived, because packages that have similar features would be merged, reducing the effects of any single feature. This varying window size could have the potential negative effect of lowering the predictive performance of the resulting classifiers, but it turns out, as subsequent chapters will show, that this is not a major problem.

To read more about the experiments that demonstrate the difficulty of bypassing these features, see "DBank: Predictive Behavioral Analysis of Recent Android Banking Trojans" by Chongyang Bai et al. and "Android Malware Detection via (Somewhat) Robust Irreversible Feature Transformations" by Qian Han et al.

Landmark-Based Features

Another way to generate features for Android apps that attackers can't easily evade relies on the concept of landmarks. Suppose you are considering buying a house. Your estimate of a fair price for the house will likely depend upon several factors, one of which might be the sales prices of certain other houses (for example, those of a similar size and age in the same area). We call these reference houses *landmarks*.

We can adopt the idea of using landmarks to define a new feature space for Android apps. Say there is a set of Android apps that includes both benign and malicious apps, and that each app has some feature vector. We can think of that feature vector as a point in the app feature space, just as we could characterize a house as a point in a housing feature space. When considering buying a house, we compare the house with similar houses; we can do the same with apps when trying to determine whether they are malicious or benign.

Selecting Landmarks

To use the landmark approach, we first select a subset of the app samples and set them as landmarks. Then we define new features for each app in the dataset by comparing them with each landmark. We suggest keeping the size of the landmark set reasonably small. For example, if there are 1 million samples in the total set of apps, we might select 1,000 landmarks. That way, adversaries will have trouble guessing the selected landmarks, making it even harder to guess the landmark-based features.

We propose three methods for selecting the set of landmarks from the sample set. The first, a naive approach, is to randomly select them. Another method is *clustering-based selection*, in which the apps are first clustered into groups. There are many well-studied algorithms for clustering, such as *k*-means clustering, *k*-median clustering, mean shift clustering, density-based spatial clustering of applications with noise (DBSCAN), expectation maximization clustering using Gaussian mixture models, and agglomerative hierarchical clustering. Each clustering algorithm has its own advantages and disadvantages. They may also perform differently due to the characteristics of the dataset.

With this approach, after clustering the apps into groups, we select one app from each group as a landmark. The basic idea is that when we group all the apps into clusters, similar apps end up in the same cluster; we can then pick one representative app from each of the clusters. Returning to our housing analogy, the houses in a cluster might have similar neighborhoods, local schools, square footages, prices, and numbers of bedrooms. When deciding whether a house is good or not, we might use one representative from each cluster as a landmark. Once we have our clusters, we can select a representative from each group in many ways. For instance, we could randomly select an app from the cluster. Alternatively, we could compute the sum of the distances of each app in the cluster to each of the other apps in the cluster, then use the app that has the smallest sum—the most "central" app in the cluster—as the landmark. (The distance between two apps can be calculated by finding the distance between their feature vectors, using a metric such as Euclidean distance or cosine distance.)

Because there are at least 6 clustering algorithms we can use and at least 2 ways of selecting a landmark app from each cluster, there are at least 12 ways of performing clustering-based landmark selection, even when disregarding the variability in hyperparameters that some of the clustering techniques use internally. In fact, there are many more ways of performing clustering-based landmark, e.g. by varying *k* in the *k*-means clustering and *k*-median clustering algorithms.

The third method, *maximum distance heuristic selection*, provides an algorithm for selecting landmarks that are scattered across the basic feature space. As input, it accepts the set of apps D and the number of landmarks N_L to select, as well as a distance function d used to evaluate the distance between two app samples based on their feature vectors. We might, for example, use well-known distance functions such as Euclidean distance, Manhattan distance, cosine distance, or Hamming distance. The algorithm is as follows:

The Max-Distance Heuristic Selection Algorithm

1. Randomly select an app from D and add it to the landmark set L'.
2. If $|L'| < N_L$, draw a random set of apps, R, from $D - L'$.
3. Choose the best landmark from R using one of the following methods:

$$\arg\max_{r \in R} \sum_{\ell \in L'} d(\ell, r)$$

$$\arg\max_{r \in R} \min_{\ell \in L'} d(\ell, r)$$

$$\arg\max_{r \in R} \text{median}_{\ell \in L'} d(\ell, r)$$

4. Add the selected landmark to L'.
5. When $|L'| = N_L$, use L' as the set of landmarks L.

It starts by randomly choosing an app from D as a landmark and adding it to the current set of selected landmarks, L' (step 1). It then iteratively adds more landmarks (steps 2 through 4). In each iteration, it randomly draws a set of apps from $D - L'$ (step 2), and then selects the app that is farthest away from the current set of landmarks in L' (step 3).

The distance can be calculated in various ways. For instance, suppose in a given iteration of the algorithm we have 3 landmarks, ℓ_1, ℓ_2, ℓ_3, and suppose $D - L'$ contains 100 landmarks, $\ell'_1, \ell'_2, \ldots, \ell'_{100}$. In this case, any one of the 100 landmarks may be added into L' as a fourth landmark. We could choose to add the landmark ℓ'_j that maximizes the distance from the candidate fourth landmark in $D - L'$ to the previously selected landmarks in L', or in other words maximizes the sum $\Sigma_{i=1}^{3} d(\ell_i, \ell'_j)$. Alternatively, we could choose the fourth landmark to be the one in $D - L'$ that maximizes either the mean distance or the median distance to the previously chosen landmarks ℓ_1, ℓ_2, ℓ_3, for example by choosing $\ell_j = \mathbf{argmax}_{\ell'_i} mean(\{d(\ell'_i, \ell_1), d(\ell'_i, \ell_2), d(\ell'_i, \ell_3)\})$. d in this algorithm is a distance function. We let $d(\ell, r)$ denote the distance between the feature vectors of two apps, ℓ and r. This step ensures that the landmark selected is sufficiently far away from the previously selected landmarks to ensure some diversity among the landmark set.

The process ends when N_L landmarks have been picked (step 5). As there are 4 distance functions and 3 possible definitions of farthest distance, we can apply this landmark selection method in at least 12 ways.

Between the three landmark selection methods we've described, there are numerous ways to select the set L of landmarks from the set D for each N_L value. However, to further confound potential adversaries, we suggest that security officers periodically use a new set of landmarks, modify the landmark selection method, or both, and then recompute landmark-based features. By doing this once every week or two, you'll keep any adversaries guessing and mount a moving target defense.

Computing Landmark-Based Features

Once we've selected landmarks, we use them to compute landmark-based features for each app sample i in set D. Here is the algorithm for generating landmark-based features:

The Landmark-Based Feature Generation Algorithm

1. Generate the set of landmarks L using S.
2. For each landmark $\ell \in$ in each sample app $i \in D$, compute $d(i, \ell)$.
3. Compute the features as follows:

$$\vec{f}_i^{lm} = \{d(i, \ell)\}_{\ell \in \mathcal{L}}$$

As input, we use the set D of Android apps with their associated feature vectors $F = \{\vec{f}_i\}_i$, the number N_L of landmarks to select, the landmark-selection method S (and its parameters, if applicable), and the distance function $d(\cdot)$.

We generate the set L of landmarks using S (step 1). Next, we iteratively compute the landmark feature vectors for each sample app i (steps 2 and 3). This process begins by computing the distance $d(i, \ell)$ of the sample i to each $\ell \in$, then constructing an N_L−dimensional landmark-based feature vector by using those distances. In other words, the first element in this vector is the distance between app i and the first landmark, the second element in this vector is the distance between app i and the second landmark, and so forth.

Figure 6-3 is a simple illustration of landmark features. It assumes that there are six samples in our set D (in practice, this number would be much larger), each with a four-dimensional API feature vector.

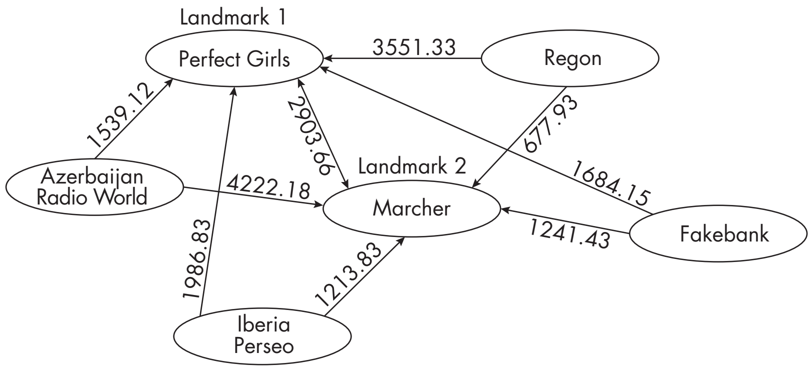


Figure 6-3: Landmark-based features with six apps, two landmarks, and the Euclidean distance function

Suppose we use the random landmark generation method to select two of the six samples, Perfect Girls and Marcher, as landmarks. We then generate landmark features using the Euclidean distance function. Here, you can see the Euclidean distance from each sample app i to each landmark. The landmark-based feature vector for, say, Regon is then (3551.33, 677.93), while that for Perfect Girls is (0, 2903.66).

Feature Clustering

Some of the features we generate might have similar relationships to the label we're attempting to predict. When this happens, we can combine those features to create a smaller, but perhaps more representative, set of new features. The approach, called *feature clustering*, first groups a set of basic features into a number of categories and then derives aggregated features from each category. We call these new features *FC features*. You can read more about this approach in "Android Malware Detection via (Somewhat) Robust Irreversible Feature Transformations" by Qian Han et al.

Generating Feature Clusters

We use the following algorithm to get FC features:

The FC Feature Generation Algorithm

1. Take a subset of samples D' from D.
2. Get the feature matrix F' for samples in D'.
3. Using *Clu*, cluster the n basic features into G groups according to column vectors $\{f_{ij}\}_{i'}$ in F'.
4. For each feature group F_g in each sample app i, associate a value with the group:
$$f_{ig}^{fc} = \oplus\{f_{ij} \mid j \in F_g\}$$
5. Perform this calculation for each sample app:
$$\vec{f}_i^{fc} = (f_{i1}^{fc}, \ldots, f_{iG}^{fc})$$

As input, it takes the set D of all sample Android apps and each of their n–dimensional basic feature vectors; the number G of clusters in which to divide the n features; the clustering algorithm used, *Clu*; and $\oplus$, the algorithm to aggregate features within one group. We can use any subset or all of the basic static and dynamic analysis features we've presented, as well as features defined by other researchers.

We extract a subset D' of sample apps from D (step 1) and use their feature values (step 2) to cluster the n features into G groups (step 3). We use a subset of D, not D itself, for three reasons: first, the dataset might be huge, and clustering the whole thing could be very expensive; second, by using a subset of samples for clustering, we make it harder for an adversary to determine how the feature clustering works; and third, when the set D is extended with the addition of more apps, we can compute the FC features of the new apps without having to rerun the algorithm and recluster basic features. Moreover, as in the case of TSGs, we can periodically update the sample used and recompute the feature clusters to keep adversaries guessing about the nature of the defenses used.

Once we've clustered the features, we take any app and use $\oplus$ to associate a single value with each cluster of features (step 4). That value could be a sum, a minimum, or a maximum of the values of the features within that cluster, or it could be a statistical quantity derived from the set, such as the median, standard deviation, variance, or entropy. We perform this action for all clusters in every app in D (step 5).

Choosing Clustering and Feature Aggregation Algorithms

We can invoke the feature clustering algorithm with many possible clustering and feature aggregation methods. For the clustering algorithm, we might use any of the six methods we mentioned in our discussion of landmark-based features or an entirely different algorithm. We can also choose from numerous possibilities for the feature aggregation algorithm, $\oplus$. Here are some options:

Product We compute the new feature as the product of elements in the set.

Mean We use the mean value of the set of values as the new feature value.

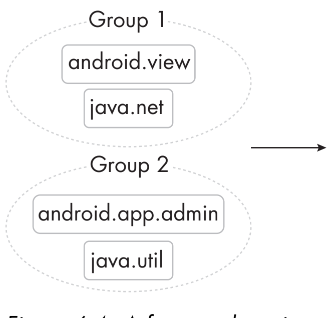
Median We use the median value of the set of values as the new feature value.

Sum We compute the new feature as the sum of elements in the set.

Weighted sum We compute the new feature value as the weighted sum of elements in the set. The weight of feature j is inversely proportional to the distance between the feature's vector and the centroid feature value of the group j_c's vector $\{f_{ij_c}\}_{i'}$, which we denote as $d(j, j_c)$. Thus, we compute the feature value as follows, where α is a parameter for normalization:

$$f_{ig}^{fc} = \alpha \sum_{j \in \mathcal{F}_g} f_{ij} \times e^{-d(j,j_c)}$$

We usually select a cluster size G that is significantly smaller than the total number of features so that this number decreases dramatically. For instance, if the basic feature vector had 100 elements, we might set G to 8. Figure 6-4 illustrates an example of feature clustering that uses sample apps and four-dimensional API features.



APK	f_{i1}^{fc}	f_{i2}^{fc}
Perfect Girls	1093.5	1376.5
Azerbaijan Radio World	1269	2146
Iberia Perseo	292.5	842.5
Regon	17.5	56.5
Marcher	302.5	240.5
Fakebank	670	722.5

Figure 6-4: A feature clustering example with two groups, four-dimensional basic API features, and averaging feature aggregation

In this example, we cluster the four apps into two groups and use the mean approach for $\oplus$. We obtain the FC features for each app shown in the table on the right.

While highly representative, FC features are hard for adversaries to guess, since generating them requires security analysts to make several choices that inject considerable uncertainty into the process and are difficult to reverse engineer. These choices include the subset of sample apps to use, the number of clusters to generate, the clustering method and its hyperparameters, and the aggregation operator $\oplus$ (along with its hyperparameters, when $\oplus$ calculates a weighted sum).

Correlation Graph–Based Feature Transformation

Another way to reduce the number of features is to use correlation graphs, which generate what we call *CG features*. This approach involves creating a fully connected graph with features as its vertices, then using concepts from social network analysis to divide these features into communities. As each community consists of similar features, we can associate one CG feature with each.

We use the following algorithm to perform correlation graph–based feature transformation:

The CG Feature Generation Algorithm

1. Take a subset D' of samples from D.
2. Get the feature matrix F' for samples in D'.
3. Compute the $n \times n$ edge weights of the correlation graph according to the column vectors of F'.
4. Get G communities with the n basic features according to the correlation graph and the community detection algorithm.
5. For each feature community C_g in each sample app i, apply the aggregation operator:

$$f_{ig}^{cg} = \oplus\{f_{ij} \mid j \in_g\}$$

6. For each sample app i, calculate its CG feature vector:

$$\vec{f}_i^{cg} = (f_{i1}^{cg}, \ldots, f_{iG}^{cg})$$

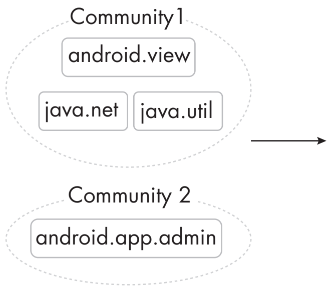
As input, it takes the set of apps D, the feature matrix F of those apps, a community detection algorithm C, the desired number of communities G, and an associative and commutative operator $\oplus$. It outputs a correlation graph with G-dimensional feature vectors for sample apps in D.

We begin by selecting a subset D' of sample apps from D (step 1) and retrieving their feature matrix F' (step 2), just as we did when calculating FC features. We then compute the *correlation* between each pair of features using the Pearson correlation coefficient (step 3). This value becomes the weight of the edge between each pair of features in the correlation graph.

Next, we apply the community detection algorithm C (step 4) to produce *G* communities. Finally, we generate the CG features for each app *D* using the features in each community and the associative and commutative feature aggregation operator $\oplus$ (steps 5 and 6).

We can define $\oplus$ in the same five ways as for feature clustering. In addition, we can select many possible community detection algorithms C, including the minimum cut method, the Girvan–Newman algorithm, modularity maximization, statistical interference, and clique-based methods. You can read more about these algorithms in the resources listed in the "Further Reading" section.

Figure 6-5 shows an example of generating correlation graph–based features. Suppose we want to group four API features into two communities, as shown on the left side of the figure. On the right side, you can see the CG features for each sample app created using the averaging feature aggregation method.



APK	f^{cg}_{i1}	f^{cg}_{i2}
Perfect Girls	1766.67	0
Azerbaijan Radio World	2276.67	0
Iberia Perseo	823.33	0
Regon	49	1
Marcher	361.33	2
Fakebank	961.67	0

Figure 6-5: Generating CG features with two communities and the averaging feature aggregation method

As with feature clustering, the use of CG features injects a great deal of uncertainty for any adversary attempting to reproduce the CG features. The CG feature generation process consists of many different choices that may end up yielding big differences in the final feature values. Adversaries will therefore have considerable difficulty in determining its real-world implementation.

Further Reading

This section lists resources you can use to further explore the topics introduced in this chapter.

To learn more about API-based features like the ones introduced in this chapter, see "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android" by Yousra Aafer et al. and "Machine Learning for Android Malware Detection Using Permission and API Calls" by Naser Peiravian and Xingquan Zhu.

To read about TSG features, consult the paper that introduced them, "DBank: Predictive Behavioral Analysis of Recent Android Banking Trojans" by Chongyang Bai et al. In addition, we mentioned that TSGs are an alternative to the many kinds of function call graphs used in other malware detection techniques:

- Dependency graphs, introduced in "Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs" by Mu Zhang et al.
- Control-flow graphs, introduced in "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps" by Steven Arzt et al. and "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models" by Enrico Mariconti et al.
- Code-property graphs, introduced in "Modeling and Discovering Vulnerabilities with Code Property Graphs" by Fabian Yamaguchi et al.

Generating the CG features introduced in this chapter requires the use of a community detection algorithm. There are many ways of defining such an algorithm:

- The minimum cut method, described in "Odd Minimum Cut-Sets and b-Matchings" by Manfred W. Padberg and M. Ram Rao
- Hierarchical clustering, described in "Hierarchical Clustering Schemes" by Stephen C. Johnson
- The Girvan–Newman algorithm, described in "Community Structure in Networks: Girvan–Newman Algorithm Improvement" by Ljiljana Despalatović et al.
- Modularity maximization, described in "Community Detection via Maximization of Modularity and Its Variants" by Mingming Chen et al.
- Statistical inference, described in Kate Calder's *Statistical Inference* (Holt, 1953)
- Clique-based methods, described in "A Maximal Clique Based Multiobjective Evolutionary Algorithm for Overlapping Community Detection" by Xuyun Wen et al.

Up Next

Whenever antivirus products detect a piece of malware, the malware's developers modify it in order to evade detection. By now, malware developers understand that antivirus companies are increasingly using machine learning. They're also well aware of the types of basic features used to detect their malware and have become adept at modifying their code to change these features to escape detection.

In this chapter, we described how to use the manual processes of static and dynamic analysis introduced in Chapters 3 and 4 to define features that machine learning algorithms can use. We then discussed two broad classes of techniques that can make life harder for malware developers. The first, based on the notion of a triadic suspicion graph, was initially used to detect Android banking trojans but can in fact be used to detect any form of malware. The second transforms the original features of Android apps into a new set of features of a different size. We described three such methods in this chapter: landmark-based transformations, feature clustering, and correlation graph–based feature transformation, all of which are resilient to reverse engineering.

However, no method is perfect at confounding hackers. To further frustrate malware developers, the techniques introduced in this chapter include layers of randomization. In addition, we recommend that organizations change their machine learning–based malware detection settings frequently, just as all users should change their passwords frequently. For instance, in the case of TSGs, defenders could update the malware and goodware samples used to generate their features and modify other parameters, such as the window size, every week. In the case of landmark-based features, defenders could periodically modify the number and identities of their landmarks. These modifications impose a relatively small cost on enterprise security officers but can reap substantial benefits.

In the next chapter, we'll apply what you've learned so far about machine learning algorithms and features to look at one important class of malware: rooting malware. This type of malware attempts to acquire root privileges on the user's device, and once it has done so, it can be hard to dislodge. As a consequence, it's essential to find characteristics of rooting malware that distinguish it from goodware.