Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware

Xiaohan Zhang[*]
Fudan University
xh_zhang@fudan.edu.cn

Yuan Zhang[*]
Fudan University
yuanxzhang@fudan.edu.cn

Ming Zhong
Fudan University
19210240059@fudan.edu.cn

Daizong Ding
Fudan University
17110240010@fudan.edu.cn

Yinzhi Cao
Johns Hopkins University
yinzhi.cao@jhu.edu

Yukun Zhang
Fudan University
16307130205@fudan.edu.cn

Mi Zhang
Fudan University
mi_zhang@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

ABSTRACT

Machine learning (ML) classifiers have been widely deployed to detect Android malware, but at the same time the application of ML classifiers also faces an emerging problem. The performance of such classifiers degrades—or called ages—significantly over time given the malware evolution. Prior works have proposed to use retraining or active learning to reverse and improve aged models. However, the underlying classifier itself is still blind, unaware of malware evolution. Unsurprisingly, such evolution-insensitive retraining or active learning comes at a price, i.e., the labeling of tens of thousands of malware samples and the cost of significant human efforts.

In this paper, we propose the first framework, called APIGraph, to enhance state-of-the-art malware classifiers with the similarity information among evolved Android malware in terms of semantically-equivalent or similar API usages, thus naturally slowing down classifier aging. Our evaluation shows that because of the slow-down of classifier aging, APIGraph saves significant amounts of human efforts required by active learning in labeling new malware samples.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; **Mobile platform security**.

KEYWORDS

Evolved Malware Detection, API Semantics, Model Aging

ACM Reference Format:
Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. 2020. Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372297.3417291

[*]co-first authors

1 INTRODUCTION

Machine learning (ML) classifiers are widely used in practice to detect Android malware [1, 3, 12, 18, 19, 29, 32, 33, 37, 49, 50, 52] and have achieved astonishing performance. Despite the success, one emerging problem of applying ML in malware detection is the evolution of malware to enhance functionalities and avoid being detected, thus leading to significant performance degradation of ML classification models over time. This problem is defined as model aging or similar concepts like time decay [39], model degradation [24], and deterioration [9] in the literature. Model aging is severe: A white paper [23] from Kaspersky in 2019 shows that the detection rate of a commercial, ML-based classifier drops drastically from almost 100% to below 80%—or even 60% under another configuration—in only three months.

Given the severeness of the aging problem, prior works have proposed to detect model aging and improve malware classifier's performance. For example, DroidOL [37] and DroidEvolver [49] keep introducing new malware samples via online learning. For another example, Transcend [21] detects early signals of model aging and retrains the model for improvement. Following Transcend, Tesseract [39] introduces active learning to choose a small set of representative evolved malware samples for improvement. However, although prior works can reverse aging and improve decayed models, the underlying model is still largely unaware of malware evolution, especially the semantics among evolved malware. Unsurprisingly, they need tens of thousands of new malware samples with labels to let the underlying model pick up the evolution, which involves a large amount of human work in labeling.

In this paper, the research problem that we study is to understand why malware evolution can degrade model performance and then enhance existing classifiers with evolution semantics to slow down aging. When aging is being slowed down, fewer new samples—and thus less human efforts in labeling—are needed to improve the classifier less frequently via either retraining, active learning, or online learning. In the light of this problem, our *key* observation is that malware samples, during evolution, often keep the same semantics but switch to a different implementation so that the evolved malware can avoid being detected by existing classifiers. For example, the original malware may send one user identifier like IMEI via HTTP requests, but the evolved one could send a different identifier such as IMSI via sockets. Semantically, they are almost the same, but the directly observed implementations are different.

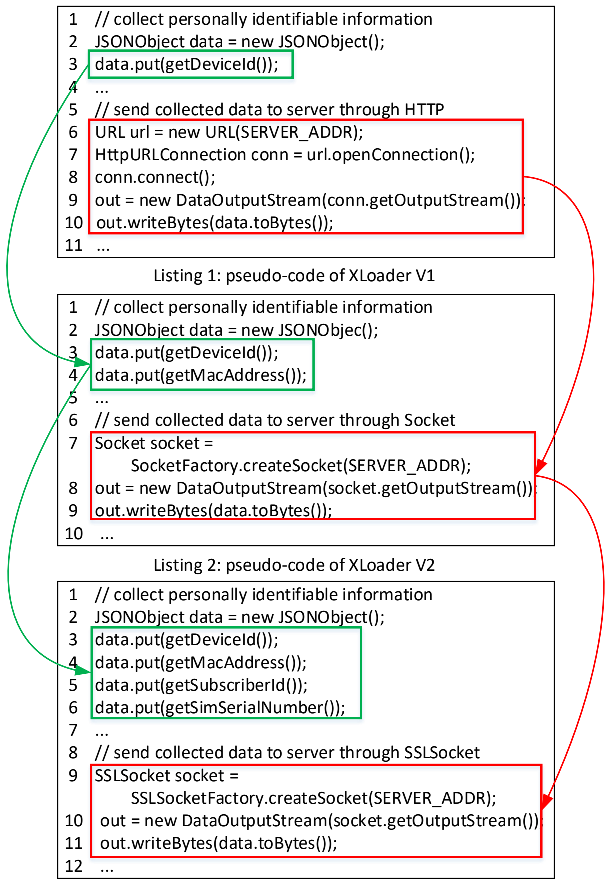
Following our observation, we propose to capture the semantic similarity during malware evolution and use the captured information to slow down the aging of malware classifiers. Our insight is that if two behaviors—e.g., the invocation of different Android APIs—are semantically similar, such similarity will also be reflected in the official Android document like API references for developers. For example, the API documents of both an HTTP request and a plain socket mention Internet access. Therefore, we can extract such common, semantic knowledge among different Android APIs and group them to be used in malware classifiers.

Specifically, we design a framework, called APIGraph, to construct a so-called relation graph of Android APIs based on information provided in and extracted from the official documents. Each node in the graph represents a key entity, such as an API, an exception or a permission; and each edge represents the relation between two entities, such as one API throwing an exception or requiring a permission. APIGraph then extracts API semantics from the relation graph by converting each API entity into an embedding and grouping similar APIs into clusters. The extracted API semantics in the format of API clusters can be further used in existing Android malware classifiers to detect evolved malware, thus slowing down aging.

We apply APIGraph upon four prior Android malware classifiers, namely MamaDroid [32], DroidEvolver [49], Drebin [3], and Drebin-DL [18], and evaluate them using a dataset created by ourselves following existing guidelines [39], which contains more than 322K Android apps ranging from 2012 to 2018. Our evaluation shows that APIGraph can significantly reduce the labeling efforts of the aforementioned four malware classifiers—i.e., ranging from 33.07% to 96.30% depending on the classifier—when combined with the active learning in Tesseract [39]. We also measure the Area Under Time (AUT), a new metric proposed by Tesseract, and show a significant slowdown of model aging with the help of APIGraph.

Contributions. This paper makes the following contributions.

- We show that although Android malware evolves over time, many semantics are still the same or similar, leaving us an opportunity to detect them after evolution.
- We propose to represent similarities of Android APIs in a relation graph and design a system, called APIGraph, to build API relation graphs and extract semantics from relation graphs.
- We build a large-scale evolved dataset spanning over seven years—the dataset is almost three times of the one used in the state-of-the-art work [39] in evaluating model aging.
- We apply the results of APIGraph, i.e., API clusters, to four state-of-the-art Android malware detectors, and show that the manual labeling efforts are significantly reduced and the aging of these models is significantly slowed down.



```
// collect personally identifiable information
JSONObject data = new JSONObject();
data.put(getDeviceId());
...
// send collected data to server through HTTP
URL url = new URL(SERVER_ADDR);
HttpURLConnection conn = url.openConnection();
conn.connect();
out = new DataOutputStream(conn.getOutputStream());
out.writeBytes(data.toBytes());
...
```

Listing 1: pseudo-code of XLoader V1

```
// collect personally identifiable information
JSONObject data = new JSONObjec();
data.put(getDeviceId());
data.put(getMacAddress());
...
// send collected data to server through Socket
Socket socket =
      SocketFactory.createSocket(SERVER_ADDR);
out = new DataOutputStream(socket.getOutputStream());
out.writeBytes(data.toBytes());
...
```

Listing 2: pseudo-code of XLoader V2

```
// collect personally identifiable information
JSONObject data = new JSONObject();
data.put(getDeviceId());
data.put(getMacAddress());
data.put(getSubscriberId());
data.put(getSimSerialNumber());
...
// send collected data to server through SSLSocket
SSLSocket socket =
      SSLSocketFactory.createSocket(SERVER_ADDR);
out = new DataOutputStream(socket.getOutputStream());
out.writeBytes(data.toBytes());
...
```

Listing 3: pseudo-code of XLoader V3

Figure 1: A motivating example to illustrate semantic similarities of different malware variations during evolution.

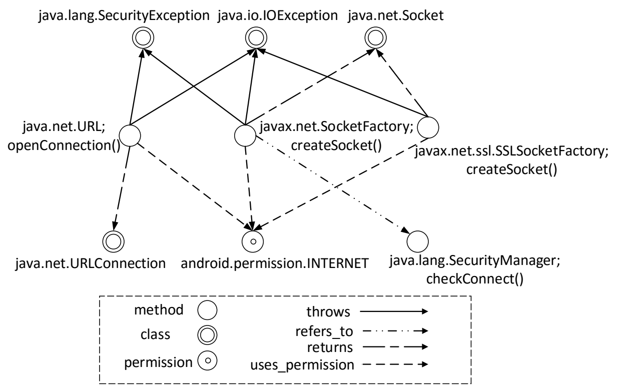
2 OVERVIEW

In this section, we start from a motivating example and then give an overview of the system architecture.

2.1 A Motivating Example

We illustrate a real-world, motivating example to explain how APIGraph captures the semantics across various malware versions during evolution. The malware example, called `XLoader`, is a spyware and banking trojan that steals personally identifiable information (PII) and financial data according to TrendMicro [34, 35]. Although `XLoader` has evolved into six different variations with large implementation changes from April 2018 until late 2019, many semantics across these variations still remain the same.

For the purpose of clear descriptions, we reverse engineered and simplified the implementation of `XLoader` into three representative code snippets (called V1, V2, and V3) as shown in Figure 1. We listed two types of semantics that are preserved across these three versions but with different implementations: (i) PII collection, and (ii) sending PII to malware server. First, the PII collection evolves from a single source in V1 to two in V2 and then multiple in V3. Specifically, V1 only collects the device ID, i.e., the IMEI, V2 adds



java.lang.SecurityException
java.io.IOException
java.net.Socket
java.net.URL; openConnection()
javax.net.SocketFactory; createSocket()
javax.net.ssl.SSLSocketFactory; createSocket()
java.net.URLConnection
android.permission.INTERNET
java.lang.SecurityManager; checkConnect()
method
class
permission
throws
refers_to
returns
uses_permission

Figure 2: An illustrative relation graph to demonstrate how APIGraph captures the semantics across different versions of XLoader in Figure 1.

the MAC address, and V3 IMSI and ICCID. Second, the malware sends PII to the malware server via three different implementations, which are an HTTP request (Lines 6–10 in V1), a plain socket connection (Lines 7–9 in V2), and an SSL socket connection (Lines 9–11 in V3).

Next, we explain how APIGraph captures the semantic similarity among three different versions of `XLoader` in terms of sending PII and thus helps ML classifiers trained with V1 to detect evolved V2 and V3. Figure 2 shows a small part of the relation graph constructed by APIGraph, which captures the interplays of Android APIs, permissions and exceptions. All three APIs—i.e., `openConnection`, `SocketFactory.createSocket`, and `ssl.SSLSocketFactory.createSocket`—throw `IOException` and use `INTERNET` permission; and two of these three APIs share more exceptions and permissions. That is, these three APIs are close enough in terms of neighborhoods in the relation graph and can be group together in a cluster. Therefore, an ML classifier, under the help of the relation graph, can capture the similarity between V2/V3 and V1 and detect V2 and V3 as a malware after the evolution.

2.2 System Architecture

Figure 3 shows the overall architecture of APIGraph, which builds on a central piece of a concept called API relation graph capturing the semantic meaning and similarities of all the Android APIs. There are two major phases of APIGraph: (i) building API relation graph, and (ii) leveraging API relation graph. First, APIGraph builds an API relation graph by collecting Android API documents related to a certain API level and extracting entities—such as APIs and permissions—and relations between those entities.

Second, APIGraph leverages the API relation graph to enhance existing malware detectors. Specifically, APIGraph converts all the entities in the relation graph into vectors using graph embedding algorithms. The insight here is that the vector difference between two entities in the embedding space reflects the semantic meaning of the relation. Therefore, APIGraph generates all the entities embeddings via solving an optimization problem so that the vector of two entities with the same relation is similar. Then, APIGraph clusters all the API entities in the embedding space to group semantically similar APIs together. Those API clusters are further used to enhance existing classifiers so that they can capture the semantic-equivalent evolution of Android malware using certain API levels during detection, thus slowing down aging.

3 DESIGN

In this section, we first define the key concept, i.e., our API relation graph, and then describe how to build and leverage this API relation graph.

3.1 Definition of API Relation Graph

An API relation graph $G = \langle E, R \rangle$ is defined as a directed graph, where E is the set of all nodes (called entities), and R is the set of all edges (called relations) between two nodes. API relation graph is heterogeneous, i.e., there are different entity and relation types as discussed below.

Entity Types. There are four types of entities in API relation graph, which are basic concepts in Android: *method*, *class*, *package* and *permission*. The former three entity types are key code elements to organize Java programs and the last one depicts the resources that an Android API needs during its execution. The four entities together provide enough capability in capturing the internal relationships among APIs.

Relation Types. We define ten relation types following a relation taxonomy provided by prior works [25, 30], which covers diverse information about an API profile. These ten types of relations, as shown in Table 1, are also summarized into five categories and described below.

- *Organization* category describes the code layout relationships between different entities. Considering the four entity types, we define *class_of* relation to connect a *class* entity with its belonging *package* entity, *function_of* relation to connect a *method* entity with its belonging *class* entity, and *inheritance* relation to connect a *class* entity with its inherited *class* entity.
- *Prototype* category describes the prototype of a *method* entity, including three types of relations: *uses_parameter*, *returns*, *throws* relations, which reflect one *method* entity may use a *class* entity as its parameter, return value, or thrown exception respectively.
- *Usage* category specifies how to use an API. We focus on two types of such relation: *conditional* relation specifies the usage of one *method* entity is on conditional of another *method* entity, e.g., one API should be used only after another API is called; *alternative* relation depicts that one *method* entity can be replaced by another *method* entity.
- *Reference* category has a *refers_to* relation that describes a general relationship between two entities. For example, the API document may refer another *method* entity when describing one *method* entity using a sentence like “see also ...”.
- *Permission* category contains the *uses_permission* relation to describe the *permission* entity that a *method* entity may require.

To build the API relation graph, we need to extract entities and relations of the above types. In the rest of this section, we first introduce the organization of Android API reference documents. Then we describe how to extract entities and relations of different types from these documents.

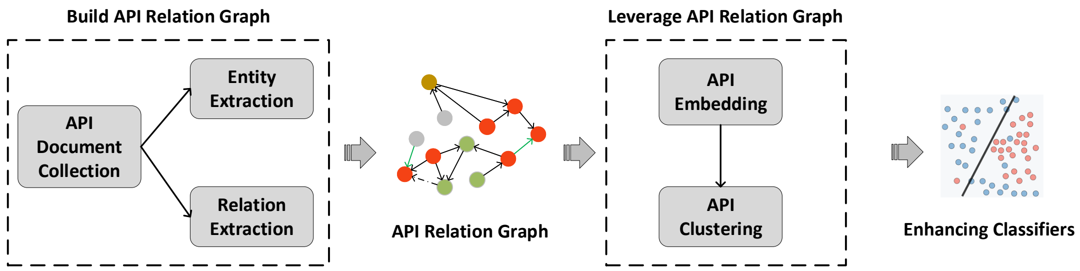


Figure 3: The overall architecture of APIGRAPH.

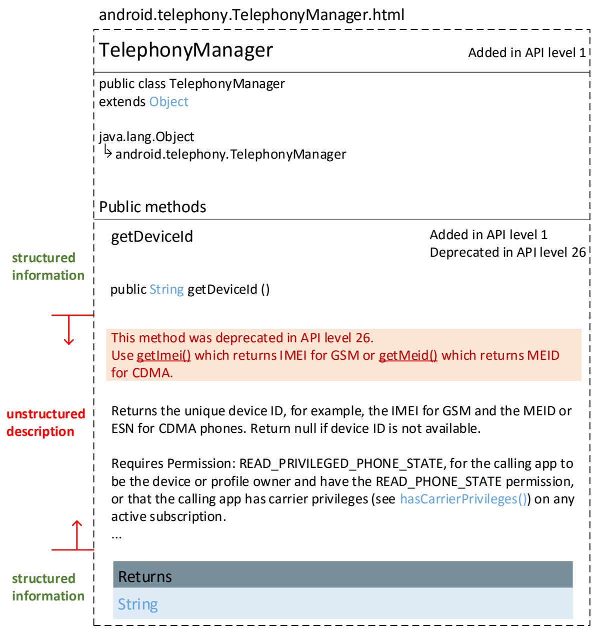
Table 1: Relation types defined in APIGRAPH.

Perspective	Relations	Entities	Examples
Organization	class_of	class→ package	*java.net.Socket* is class_of *java.net*
	function_of	method→ class	*BluetoothDevice.getAddress()* is function_of *android.bluetooth.BluetoothDevice*
	inheritance	class → class	*javax.net.ssl.SSLSocketFactory* inheritance *javax.net.SocketFactory*
Prototype	uses_parameter	method→class	*javax.net.SocketFactory.createSocket()* uses_parameter *java.net.INetAddress*
	returns	method→class	*java.net.Socket.getInputStream()* returns *java.io.InputStream*
	throws	method→class	*LocationManager.requestLocationUpdates()* throws *java.lang.SecurityException*
Usage	conditional	method→method	"This method should be called after ...", "... is called when ..."
	alternative	method→method	"This method is deprecated, use ... instead", "is replaced by ..."
Reference	refers_to	method→ method method→class	"Please refer to ...", "see also ..."
Permission	uses_permission	method→permission	"requires INTERNET permission"

3.2 API Document Collection

APIGRAPH downloads API reference documents for all platform APIs and support libraries from the official website[1]. Each Android version has a corresponding API level, e.g. Android 10 has API level 29. APIGRAPH crawls the documents for API level 14 to 29, which correspond to Android 4.0 to Android 10 and they are the major active Android versions at present.

Android API reference documents are organized hierarchically. From the top level to the bottom level, there are packages, classes, and methods. The API documentation is given at the level of class. There is a single HTML file for each class to describe the basic class hierarchy information and also detailed documentation for all methods in this class. Figure 4 shows an example of the documentation for `android.telephony.TelephonyManager` and one method `getDeviceId()` of it. The documentation can be separated into two parts: 1) structured information including the class profile and the prototype, return value and thrown exceptions of a method, and 2) unstructured descriptions in the format of several paragraphs of text, which describe the functionality, requirements and directives of the API.



android.telephony.TelephonyManager.html

TelephonyManager Added in API level 1

public class TelephonyManager
extends Object

java.lang.Object
↳ android.telephony.TelephonyManager

Public methods

getDeviceId Added in API level 1
Deprecated in API level 26

public String getDeviceId ()

This method was deprecated in API level 26.
Use getImei() which returns IMEI for GSM or getMeid() which returns MEID for CDMA.

Returns the unique device ID, for example, the IMEI for GSM and the MEID or ESN for CDMA phones. Return null if device ID is not available.

Requires Permission: READ_PRIVILEGED_PHONE_STATE, for the calling app to be the device or profile owner and have the READ_PHONE_STATE permission, or that the calling app has carrier privileges (see hasCarrierPrivileges()) on any active subscription.
...

Returns
String

Figure 4: An example API document for *android.telephony.TelephonyManager*.

3.3 Entity Extraction

There are four entity types in our API relation graph. We extract these entity types from the documents in the following ways:

- First, since the API document is organized in classes, APIGRAPH extracts a class entity from every per-class document file. As shown in Figure 4, the name of a class is described with structured texts.
- Second, APIGRAPH extracts package entities by splitting the package names from the full class name.
- Third, APIGRAPH parses per-class document files into Document Object Model (DOM) and then extracts method entities belonging to a class.
- Fourth, APIGRAPH parses the manifest file[2] that lists all the permissions to extract permission entities.

[1]https://developer.android.com/reference

[2]https://developer.android.com/reference/android/Manifest.permission

Table 2: Templates to extract 4 relation types, where "ENT" represents an entity.

Relation Type	# of Templates	Example Templates
conditional	186	"call ENT before ENT be call", "before ENT return", "if ENT fail", "wait for ENT"
alternative	22	"replace by ENT", "use ENT instead", "be deprecate. use ENT"
refers_to	5	"see also ENT", "see ENT", "query ENT", "refer to ENT"
uses_permission	4	"require permission ENT", "require ENT permission", "be grant ENT permission"

3.4 Relation Extraction

As stated, since some relations, like *class_of* relation, are organized in well-structured HTML elements, and some, like *refers_to* relation, are embedded in unstructured texts, we adopt two methods to extract relations from API documents.

3.4.1 Relation Parsing from Structured Texts.

According to the relation types defined in Table 1, six relations are depicted structurally in the documents. APIGraph extracts these kinds of relations by direct document parsing. Here are the details. First, APIGraph extracts *function_of* and *class_of* relations during the extraction of *class*, *method* and *package* entities. Second, APIGraph extracts *inheritance* relations from the class profile part in the per-class document file. Lastly, APIGraph extracts *uses_parameter*, *returns*, and *throws* relations at the prototype part for each method.

3.4.2 Template-based Relation Matching from Unstructured Texts.

APIGraph extracts four types of relations, i.e., *conditional*, *alternative*, *refers_to* and *uses_permission*, using a template-based relation matching method with the help of NLP (Natural Language Processing) techniques. Note that APIGraph also extracts *uses_permission* relations from two API-permission mappings generated by existing works [4, 5] to complement the relations extracted from API documents, because such information in the Android API documents may be incomplete. In general, there are three steps in template-based relation extraction: (i) manual formation of matching templates, (ii) iterative expansion of template set, and (iii) NLP-enhanced template matching.

Manual Formulation of Matching Templates. In this step, we manually examine 1% of API documents to investigate the patterns that are used to describe the relations. Table 2 gives several example templates in regular expression format for each kind of relation that are manually formulated to match relations from unstructured texts. For example, the template "see also ENT" matches a *refers_to* relation between the current *method* entity and the *ENT* entity.

Iteratively Expansion of Template Set. In this step, APIGraph adopts a semi-automated strategy to iteratively formulate templates for relation matching. There are three sub-steps in this process, as described below:

- First, we randomly select 1% of APIs and collect their documents.
- Second, we use the existing template set to extract relations from these documents with the help of NLP techniques (explained in the following paragraph).
- Third, after the matching, we manually check whether there are relations not captured by existing template set. If the answer is yes, we manually formulate the templates for them and repeat from the first step. Otherwise, APIGraph finishes formulating templates to extract relations for all APIs.

Guided by the above process, the template set converges after manually looking into 5% of all API documents. Finally, we summarize 217 templates for *conditional*, *alternative*, *refers_to* and *uses_permission* relations. Table 2 presents the template number for each relation. The whole template construction process takes two security experts around three days. Note that Android documentation is stable over time. For example, only 1.4% (834) APIs are added and 1.6% (989) APIs change their descriptions from API_level 28 to 29, and none of the newly added or changed descriptions need additional templates.

NLP-enhanced Template Matching. APIGraph matches templates against unstructured API documents via two steps. First, APIGraph splits paragraphs into sentences and then preprocess each sentence via the following methods:

- *Stemming.* APIGraph reduces each word to its base form, for example, "requires" and "required" are stemmed to "require".
- *Co-reference Resolution.* APIGraph adopts declaration-based co-reference resolution [25] to resolve all the pronouns to the underlying entity. For example, "This method" in a sentence "This method requires permission INTERNET" is resolved to the method the sentence belongs to.
- *Entity Name Normalization.* APIGraph replaces all polymorphic names with their exact values via following the hyperlinks to their original definition so that APIGraph normalizes the representation of entities. For example, the name *android.Manifest.permission.INTERNET* and its constant value "android.permission.INTERNET" are both used in documents: APIGraph replaces the former with the latter.

Second, APIGraph matches all the templates against each preprocessed sentence in the API descriptions. If a match against a template is found, APIGraph then extracts relations from the sentences as specified by the template. If a sentence can not be matched with any template, APIGraph will drop the sentence.

3.5 Leveraging API Relation Graph

To leverage API relation graph, APIGraph will convert each API in the relation graph into an embedding representation and then group those embeddings into clusters. The concept of API embedding, inspired by word embedding [36] and graph embedding, is to convert each API in the relation graph to a vector, which represents its semantic meanings. Our conversion algorithm (see Algorithm 1), leveraging a prior algorithm called TransE [8] and fitting TransE into our relation graph problem, is described below:

(1) APIGraph extracts permission entities and adds new relations based on common permissions (Lines 3–5). The intuition here is that permissions in Android preserve semantics and APIGraph pays more attention to permissions.

Algorithm 1 API Embedding and Clustering

```
Input: Relation graph $G = \langle E, R \rangle$, learning rate $\lambda$, embedding size
    $k$, cluster size $C$.
 1: Set triples $S = \emptyset$                                   ▷ Form Training Set
 2: Add existing relations to triples $S$
 3: for each permission entity do
 4:     for each pair $h, t \in E$ that use this permission do
 5:         Add $(h, r_{\text{use\_the\_same\_permission}}, t)$ to triples $S$
 6: for each entity $e \in E$ do                                  ▷ Vector Initialization
 7:     Assign $e$ with a vector $l_e \in \mathbb{R}^k$
 8: for each relation $r \in R$ do
 9:     Assign $r$ with a vector $l_r \in \mathbb{R}^k$
10: while True do                                                 ▷ Train Embeddings
11:     for triple $(h, r, t) \in S$ do
12:         Minimize the following loss function:
                $\ell = \|l_h + l_r - l_t\|_2^2$
13:         Update $l_h$ by gradient descent:
                $l_h = l_h + \lambda \cdot \frac{\partial \ell}{\partial l_h}$
14:         Update $l_r, l_t, l_{t'}$ with gradient descent similarly
15:     if embeddings do not change then
16:         break
17: Collect embeddings of method entities                         ▷ Cluster APIs
18: Use k-Means algorithm to find $C$ clusters
```

(2) APIGraph embeds each API entity $e \in E$ (Lines 6–7) and each relation $r \in R$ (Lines 8–9) with vector $l_e, l_r \in \mathbb{R}^k$ respectively.
(3) APIGraph applies TransE algorithm (Lines 10–14) to minimize $\|l_h + l_r - l_t\|_2^2$ for each triple (h, r, t) in triples set S where h and t are entities and r is a relation. The intuition here is that if two head entities h_1, h_2 have the same relation with a common tail entity, their embeddings l_{h_1}, l_{h_2} should be close.
(4) APIGraph clusters API embeddings into different groups using k-Means and determines the cluster number via the Elbow method [43].

After APIGraph successfully clusters APIs, APIGraph adopts clusters, particularly the embedding of each cluster's center, to represent the semantics of independent APIs in the cluster.

4 API RELATION GRAPH RESULTS AND EXPERIMENTAL SETUP

In this section, we describe some statistics of the generated API relation graph, the dataset used in the evaluation and existing ML classifiers used in our experiment.

4.1 Statistics of API Relation Graph

Implementation. Our prototype of APIGraph contains 1,627 lines of Python code, including Android API reference document collection and parsing, relation graph building, and embedding generation and clustering. Specifically, we use spaCy [42] (a Python NLP toolkit) to perform sentence splitting, stemming and co-reference resolution, and our API embedding and clustering is built with TensorFlow [44] and sklearn [41] respectively. Following the Elbow method [43], we choose 2,000 as the total cluster number.

The results of API relation graph generated by APIGraph are described in terms of entities and relations. Specifically, we use API level 29 as an example. Table 4 shows the extracted entities: There are 67,209 entities, including 59,125 methods, 7,368 classes, 446 packages, and 270 permissions. Note that different API levels have different numbers of entities as API evolves over time. Table 4 lists the number of relations extracted for each type: There are 121,345 extracted relations among these entities.

Table 3: Extracted entities for Android API level 29.

Entity Type	Count
method	59,125
class	7,368
package	446
permission	270

Table 4: Extracted relations for Android API level 29.

Relation Type	Count	Relation Type	Count
function_of	59,125	throws	8,310
class_of	7,368	alternative	1,264
inheritance	3,755	conditional	5,990
uses_parameter	14,528	refers_to	10,859
returns	5,113	uses_permission	5,033

4.2 Dataset

Our dataset, spanning over seven years, contains 322,594 Android apps, i.e., 32,089 malicious and 290,505 benign as shown in Table 5. The dataset—following criteria documented by Tesseract [39]—has two important properties: *temporal consistency* and *spatial consistency.* The former ensures that data samples are ordered based on their appearance and almost evenly distributed over seven years; the latter that the ratio of malware is close to the percentage of malware in the real-world, which is 10% according to Tesseract for Android. Note that our dataset is almost two times larger than the one used in state-of-the-art like Tesseract. Here is how we construct this dataset.

- *Step-1: Initial malware selection and validation.* We downloaded all given Android malware from three open repositories, including VirusShare [45], VirusTotal [46][3], and the AMD dataset [27, 48]. These are three largest open-source dataset at the time we write our paper and they together contain 109,897 unique malware. We then feed all samples to VirusTotal and only keep 109,770 malware samples that are reported by at least 15[4] anti-virus (AV) engines.
- *Step-2: Initial selection and validation of benign apps.* We downloaded 1,060,000 Google Play apps with the help of AndroZoo [13]. Again, we feed all the apps to VirusTotal and only keep 1,033,073 that are reported as benign by all the AVs from VirusTotal.

[3]VirusTotal provides a set of malicious samples for academic usage at request.
[4]We follow a most recent work [49] to choose 15 as the threshold.

Table 5: Evaluation dataset. This dataset contains 322,594 apps from 2012 to 2018. For each month, the malware percentage 10%. When there are enough apps available, most months contain about 5K apps to be representative and effective for evaluation.

App \ Year	2012	2013	2014	2015	2016	2017	2018	ALL
Malicious (M)	3,066	4,871	5,871	5,797	5,651	2,620	4,213	32,089
Benign (B)	27,613	43,873	52,843	52,173	50,859	24,930	38,214	290,505
M+B	30,679	48,744	58,714	57,970	56,510	32,300	38,025	322,594
M/(M+B)	10%	10%	10%	10%	10%	10%	10%	10%

Table 6: Android malware classifiers in the evaluation. Note that DroidEvolver uses a model pool that contains 5 linear online learning algorithms.

Classifier	API feature format	Algorithm
MamaDroid [32]	Markov Chain of API Calls	Random Forest
DroidEvolver [49]	API Occurrence	Model Pool
Drebin [3]	Selected API Occurrence	SVM
Drebin-DL [18]	Selected API Occurrence	DNN

- *Step-3: Final dataset construction.* We order all the samples according to their appearance timestamps and select a subset for each month over seven years. Specifically, we select at most 500 malware samples for a month: If the number in that month is less than 500, we select all; if more than 500, we randomly select 500. We also randomly select benign apps that are nine times of malware in a month.

4.3 Candidate Classifiers and Enhancement with APIGraph

We describe four state-of-the-art, representative malware classifiers used in the evaluation and list them in Table 6. We choose all three classifiers used in Tesseract [39] together with a recent, state-of-the-art work, namely DroidEvolver [49], which delays classifier aging via a model pool. These four classifiers span over different machine learning algorithms and their usages of APIs also differ, but they all face the aging problem. Particularly, the three classifiers used in Tesseract do not update themselves; although DroidEvolver updates itself via online learning based on the majority voting of five models, the majority can age as well and such errors may propagate to all the models.

Now let us describe these four classifiers in detail and how APIGraph enhances these works by slowing down the aging process. It is worth noting that our enhancement of classifiers depends on the appearance year of the target apps—that is, if the target apps of the classifiers are from the year 2012, our enhancement will be using the API relation graph of API Level 18, because this is the latest API level in 2012.

- MamaDroid [32] extracts API call pairs (i.e. caller and callee) and then abstracts them into package call pairs.[5] Next, MamaDroid builds a Markov chain to model the transition between different packages, and the transition probabilities between packages are used as the feature vector for the app in a learning algorithm. We get the entire source code of MamaDroid and use the same configuration as its paper. APIGraph replaces each API call pair used in MamaDroid's implementation with API cluster pair and then uses such pairs in the Markov chain.
- DroidEvolver [49] finds all used APIs in an app via static analysis and then builds a binary vector of API occurrence as the feature vector for the app. After that, DroidEvolver maintains a model pool of five linear online learning algorithms to classify an app using a weighted voting algorithm. When some model in the pool aged, it will be updated incrementally based on the results of other un-aged models. We get the source code of DroidEvolver and we contact the authors to make sure our experiments are conducted consistently to their paper. In our enhancement, APIGraph replaces the binary vector of API occurrence with the one of API cluster occurrence.
- Drebin [3] gathers a wide range of features such as used hardware, API calls, permissions, and network addresses for an SVM-based classifier. In terms of the API feature, Drebin considers a selected set of restricted and suspicious APIs that can access to critical and sensitive data or resources. We implement Drebin by strictly following the detailed description and configuration in the paper. APIGraph also replaces the binary vector of API occurrence in the aforementioned subset with the one of API cluster occurrence for enhancement.
- Drebin-DL [18] uses the same feature set as Drebin but adopts Deep Neural Networks (DNN) as the algorithm to do classification. We also follow prior work [18] to implement Drebin-DL. The enhancement of Drebin-DL with APIGraph is the same as Drebin.

5 EVALUATION

In this section, we evaluate the effectiveness of APIGraph in enhancing state-of-the-art classifiers as well as in capturing the semantic similarity among Android APIs. Specifically, our evaluation answers the following four research questions.

- **RQ1: Model Maintainability Analysis.** How many human labeling efforts does APIGraph save in maintaining the high-performance of a malware classifier? (see §5.1)
- **RQ2: Model Sustainability Analysis.** How effective is APIGraph in slowing classifier aging? (see §5.2)
- **RQ3: Feature Space Stability Analysis.** How effective is APIGraph in capturing similarity among evolved malware from the same family? (see §5.3)
- **RQ4: API Closeness Analysis.** How close are APIs in clusters grouped by APIGraph? (see §5.4)

[5]MamaDroid also provides "family mode" where calls to APIs are abstracted to calls to families, but its authors prove that "package mode" is much better, so in this paper we only consider MamaDroid in package mode, as is done by previous works [9, 39].

5.1 RQ1: Model Maintainability Analysis

The purpose of this research question is to find out how many human efforts APIGraph can save while maintaining a high performance classifier. Specifically, we compare the amount of human efforts needed for active learning in maintaining both the original and the enhanced classifiers. Let us look at some details. First, the comparison adopts two metrics, which are (i) the number of malware to label, and (ii) the retraining frequency. Second, the active learning is implemented with *uncertain sampling* [39] to actively select the most uncertain predictions to label. We further adopt two settings for the active learning, which are a minimum F_1 score for introducing new samples and a fixed new sample ratio.

5.1.1 Active learning with fixed retrain thresholds. In the first setting, when the F_1 score of a classifier falls below a low threshold T_l, active learning is used to select the most 1% uncertain samples to retrain the classifier, and then gradually increase the percentage by 1% until the F_1 score reaches another higher threshold T_h. In the experiment, the classifier to start is trained on all the apps in 2012. We then adopt the aforementioned criterion to apply active learning from Jan 2013 to Dec 2018, and observe the number of malware to label and the retraining frequency.

Results: Table 7 shows the number of malware to label and the retraining frequency from 2013 to 2018 (T_l = 0.8, T_h = 0.9). APIGraph can save the amount of samples to label by 33.07%, 37.82%, 96.30% and 67.29% respectively for MamaDroid, DroidEvolver, Drebin, and Drebin-DL and the retrain frequency is reduced as well. There are three things worth noting here. First, neither Drebin and Drebin-DL are aware of model aging—thus, it is unsurprising that APIGraph can save a lot of human efforts. Second, although DroidEvolver is aware of model aging and tries to improve the model via online learning, APIGraph can still save a significant amount of human efforts. The reason is that the majority results of DroidEvolver may also make mistakes, which leads to a propagation of such mistakes to other unaged models. Lastly, Drebin, after combined with APIGraph, requires the least number of samples to label. This is interesting because although the performance of Drebin-DL is better than Drebin, Drebin with a simpler ML algorithm is easier to maintain.

We also expand the samples to label and retrain times into both cumulative and monthly-distribution numbers and show them in Figure 5. One interesting phenomenon is that the number of labeled samples for DroidEvolver and Drebin-DL stays almost the same for many months, but then suddenly increases a lot especially without the help of APIGraph. The reason is that DroidEvolver and Drebin-DL have some capabilities, to a limited degree, of capturing malware evolution, but once they do not capture one type of evolution, the consequence is catastrophic, especially for DroidEvolver. It is because DroidEvolver will propagate false evolution information to other models in the pool, leading to a false synchronization.

5.1.2 Active learning with varied learning ratios. The second active learning setting is to fix the ratio of newly introduced apps each month as 1%, 2.5%, 5%, 10%, and 50% and test the AUT(F_1, 12m) for each classifier. Similarly, we train a classifier with apps from 2012, and test the classifier month by month from Jan 2013 to Dec 2018. Note that AUT is a metric proposed by Tesseract [39], which defines the area under the curve in each figure to represent the model's sustainability as shown in Equation 1.

Table 7: [RQ1] A summarization of Figure 5 on retrain times and the number of labeled samples for active learning with fixed retrain thresholds (T_l = 0.8, T_h = 0.9).

	retrain times			# labeled samples		
	w/o [1]	w/ [2]	Improves	w/o	w/	Improves
MamaDroid	45	35	22.22%	22,411	14,999	33.07%
DroidEvolver	19	15	21.05%	20,767	12,913	37.82%
Drebin	56	13	76.79%	167,005	6,173	96.30%
Drebin-DL	26	16	38.46%	28,408	9,292	67.29%

[1] w/o denotes the classifier without APIGraph, i.e. the original classifier.
[2] w/ denotes the classifier enhanced with APIGraph.

Table 8: [RQ1] AUT(F_1, 12m) of original (w/o) and enhanced (w/) classifiers with different active learning ratios.

AL ratio	MamaDroid		DroidEvolver		Drebin		Drebin-DL	
	w/o [1]	w/ [2]	w/o	w/	w/o	w/	w/o	w/
1%	0.527	0.637	0.616	0.777	0.692	0.858	0.718	0.749
2.5%	0.619	0.712	0.693	0.840	0.745	0.878	0.766	0.811
5%	0.739	0.838	0.703	0.851	0.767	0.887	0.813	0.841
10%	0.798	0.852	0.749	0.866	0.774	0.895	0.842	0.875
50%	0.809	0.865	0.773	0.888	0.799	0.908	0.887	0.922

[1] w/o denotes the classifier without APIGraph, i.e. the original classifier.
[2] w/ denotes the classifier enhanced with APIGraph.

$$AUT(f, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{[f(k+1) + f(k)]}{2} \tag{1}$$

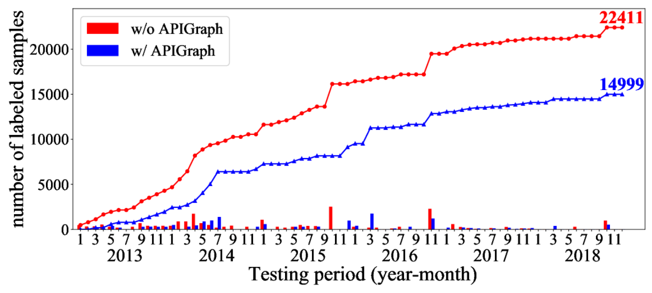
where f is the performance metric (e.g. F_1 score, Precision, Recall, etc.), N is the number of test slots, and $f(k)$ is performance metric evaluated at the time k, and in our case the final metric is AUT(F_1, 12m). An AUT metric that is closer to 1 means better performance over time.

Results: We show the results in Table 8 for the four evaluated classifiers before and after applying APIGraph. There are two things worth noting here. First, the AUT with APIGraph of each classifier is higher than the one without APIGraph. This demonstrates that APIGraph can indeed slow model aging across four different classifiers no matter they are evolution-aware or not. Second, the aging slowdown of a model enhanced with APIGraph is significant: For example, after enhancing DroidEvolver, retraining with only 1% of apps can achieve even better performance than retraining with 50% of apps for the original classifier without enhancement.

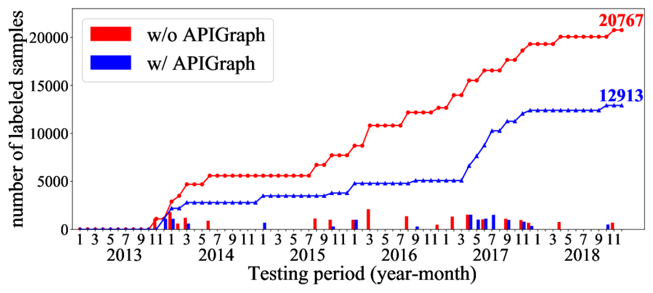
Summary: APIGraph significantly reduces (i) the number of manually-labeled samples and (ii) retrain frequency when maintaining four different, high-performance malware classifiers.

5.2 RQ2: Model Sustainability Analysis

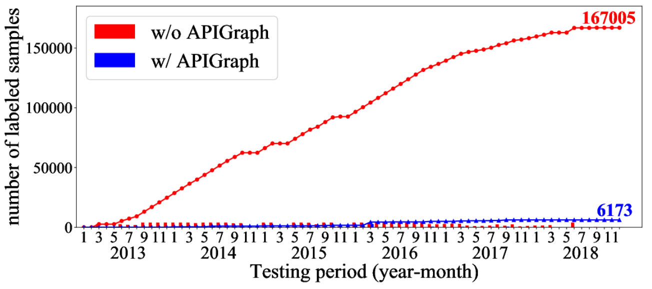
In this research question, we measure the performance of existing Android malware classifiers with and without the help of APIGraph



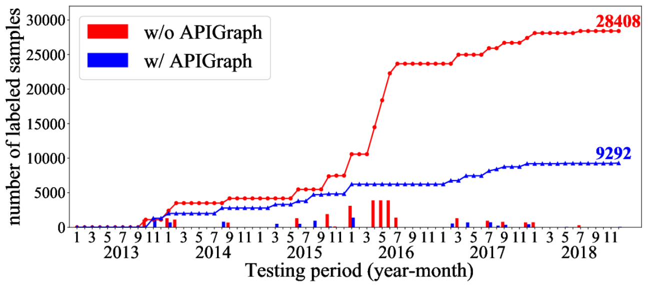
(a) The efforts in sample labeling for MamaDroid



(b) The efforts in sample labeling for DroidEvolver

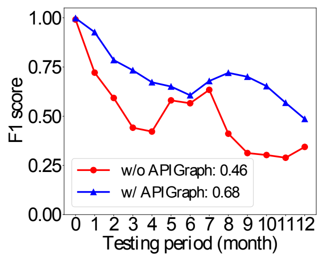


(c) The efforts in sample labeling for Drebin

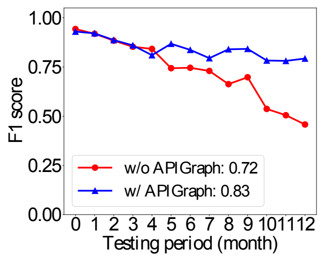


(d) The efforts in sample labeling for Drebin-DL

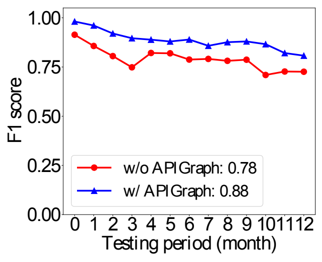
Figure 5: [RQ1] The number of malware samples to label using active learning with fixed retrain thresholds (T_l = 0.8, T_h = 0.9). All the evaluated classifiers are trained using apps from 2012, and tested using apps from 2013–2018. The bar of a month shows the number of new samples to be labeled in that month to retrain the classifier.



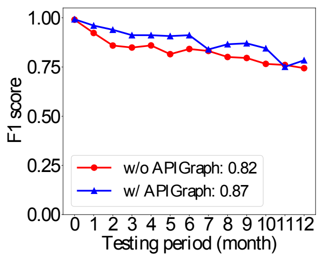
(a) MamaDroid



(b) DroidEvolver



(c) Drebin



(d) Drebin-DL

Figure 6: [RQ2] AUT(F_1, 12m) of evaluated classifiers before and after leveraging API relation graph. Each classifier is trained on 2012 and tested on 12 months of 2013. Note that month 0 indicates the time when the classifier is initially trained.

to understand the capability of APIGraph in slowing down model aging. Our experiment setup is as follows. We train a classifier on a particular year (say 2012), and test its performance on 12 months of the next year (say 2013), and then also calculate the AUT. Note that we only test the performance of a classifier over a year because many classifiers have already become unusable after one year.

Results: Table 9 shows the AUT(F_1, 12m) value of four classifiers tested from 2013 to 2018 as well as the average. One important observation is that the average AUT values improve 19.2%, 19.6%, 15.6%, 8.7% respectively for the four classifiers, which indicates that APIGraph is capable of slowing down model aging. The AUT values across different years are very similar, showing that malware keeps evolving back to 2013 until very recently in 2018.

In Figure 6, we also break down the results into months and show the F_1 score of four classifiers in 2013 when trained with data in 2012. We observe that the performance of Drebin-DL and Drebin are the best among all four classifiers in terms of aging: Specifically, the F_1 score only drops from close to 1 to above 0.8. This is probably because Drebin adopts a selected subset of APIs, which has some capability in capturing malware evolution.

Summary: APIGraph significantly enhances the sustainability of existing Android malware classifiers under evolved malware samples.

5.3 RQ3: Feature Space Stability Analysis

In this research question, we measure the feature space stability of evolved Android malware from the same family to show that

Table 9: [RQ2] AUT(F_1, 12m) of evaluated classifiers before and after leveraging API relation graph. For each testing year, the classifiers are trained on the previous year.

Testing Years	MamaDroid		DroidEvolver		Drebin		Drebin-DL	
	w/o [1]	w/ [2]	w/o	w/	w/o	w/	w/o	w/
2013	0.462	0.680	0.717	0.833	0.779	0.878	0.819	0.875
2014	0.456	0.637	0.712	0.791	0.734	0.859	0.816	0.866
2015	0.726	0.789	0.840	0.890	0.759	0.886	0.829	0.878
2016	0.718	0.814	0.718	0.875	0.666	0.869	0.706	0.916
2017	0.635	0.704	0.605	0.908	0.767	0.844	0.793	0.797
2018	0.765	0.861	0.811	0.969	0.794	0.865	0.828	0.874
Average	0.627	0.748	0.734	0.877	0.750	0.867	0.799	0.868
Improves	19.2%		19.6%		15.6%		8.7%	

[1] w/o denotes the classifier without APIGraph, i.e. the original classifier.
[2] w/ denotes the classifier enhanced with APIGraph.

APIGraph can capture semantic similarities. Here is how we obtain the family information for Android malware, which involves three steps:

- Step-1: Labeling via Euphony [20]. In this step, we use a malware labeling tool named Euphony to label the family information of all the collected 109,770 malware samples (as described in § 4.2). Euphony is capable of linking different family label aliases from all AV engines on VirusTotal. For example, Euphony can link a family label "boxersms" from one AV with its alias "boxer" from another. The output of Euphony is a list of (l, s) pairs, where l is a family label and s is the number of AVs that support this label.
- Step-2: Selection of malware with reliable labels. In this step, we choose a subset of malware with reliable labels recognized by most of the AVs. Specifically, we require that the most popular family label of a malware sample is recognized by at least 50% AVs and the second popular label is recognized by at most 10% AVs. That is, we choose the malware with dominant family labels in our study, which leads to 101,360 malware labeled with family information, covering 1,120 families.
- Step-3: Selection of top 30 malware family. In this step, we choose top 30 families that have the most number of labeled samples so that each family has enough samples for evaluation. As a result, we have 75,625 (74.61%) apps in this experiment and every family has more than 500 apps (except the last one). The top 30 families, as well as the number of their samples are listed in Table 10 of Appendix A.

Here is our evaluation methodology. We first sort all the malware samples in one family by their appearing time and then divide them into 10 groups with 10% samples of this family. The appearing time of all samples in one group is strictly ahead of the one of all samples in the next. Next we adopt static analysis with the help of *apktool* [2] to disassemble malware code and obtain API features. Lastly, we calculate a feature stability score of every two adjacent groups using Jaccard similarity (see Equation 2).

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

where A and B is the set of used features for two adjacent groups. This score function is capable of reflecting how stable the features evolve between groups.

Results: Figure 7 shows the distribution of feature stability scores for each malware family with API and API clusters as features. One important observation of the figure is that the feature stability score of all families with API clusters as features is very close to 1 and much higher than the one with API as feature directly. This observation explains that APIGraph can capture malware evolution as malware developers tend to use semantically similar APIs to implement the same or similar functionalities.

We also show the breakdown of feature stability scores for four specific malware families in Figure 8. The feature stability score with API clusters as features is almost flat without much decrease over time; by contrast, the feature stability score with independent APIs not only is low (near 0.75), but also decreases over time (sometimes to a very low value like 0.3). This, from another angle, also shows that APIGraph can capture malware evolution over time.

Summary: APIGraph successfully captures semantic similarity among evolved malware samples in a family.

5.4 RQ4: API Closeness Analysis

In this research question, we measure the closeness of APIs in the same cluster to demonstrate the effectiveness of APIGraph. Particularly, we use t-SNE [31] to project all the API embeddings into a two-dimensional space and visualize them. Figure 9 shows a subgraph of the visualization, in which those APIs in our motivating example (Figure 1) are clearly separated into different clusters. For example, PII-related APIs, such as *getDeviceId()*, *getSubscriberId()* are close to each other; and network-related APIs, such as those from "java.net", "javax.net", "android.net.Network", are also close. It is worth noting that APIs in the package "java.lang" can be clearly separated into two groups: one containing security-sensitive APIs for process management and system command execution, and the other one containing those Java built-in data structure APIs, such as *java.lang.Long.compare()*. This fact demonstrates that a simple package-level API clustering method, like that adopted by MamaDroid, is inaccurate in capturing semantics information.

Summary: Semantically-close APIs are grouped in the same or close cluster in the embedding space by APIGraph.

6 DISCUSSION

API Semantics from Non-official Documents. API semantics exist and can be extracted from many different places: official Android API documents are the main source, but other resources, such as the API tutorial and developer guides [25], can also provide semantics. We choose the Android API documents as the target because they are official and contain the most useful information. Furthermore, it will be hard for an adversary, e.g., a malware developer, to pollute official API documents and influence the performance of APIGraph.

Relation/Entity Types. We defined four entity and ten relation types in API relation graph as a start, which can be extended in the future to include more types. We believe that current types of

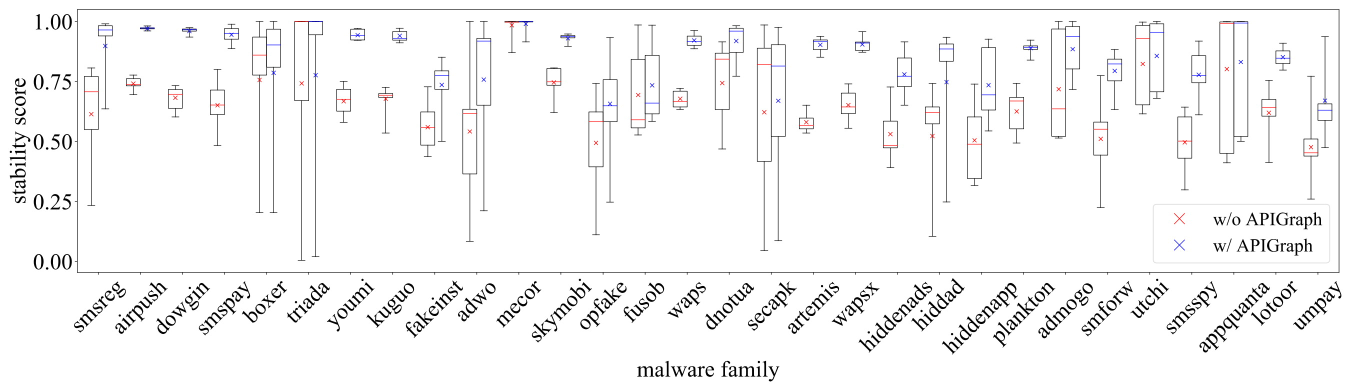


Figure 7: [RQ3] The distribution of feature stability scores for every top 30 malware family, when considering APIs as features and API clusters as features.

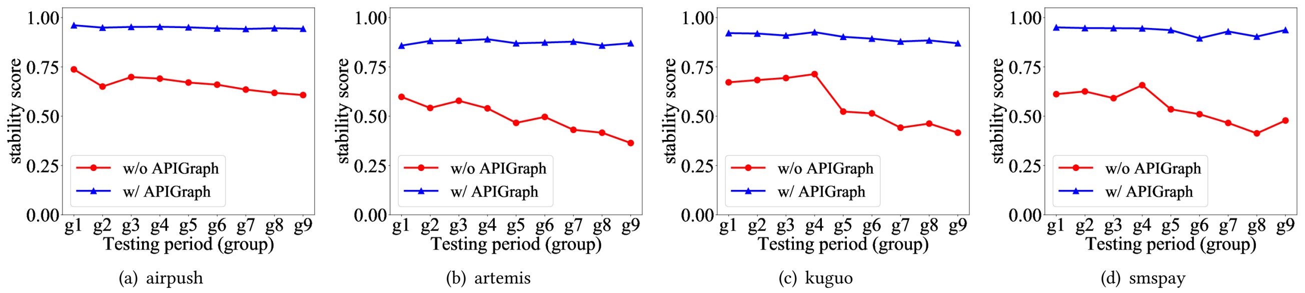
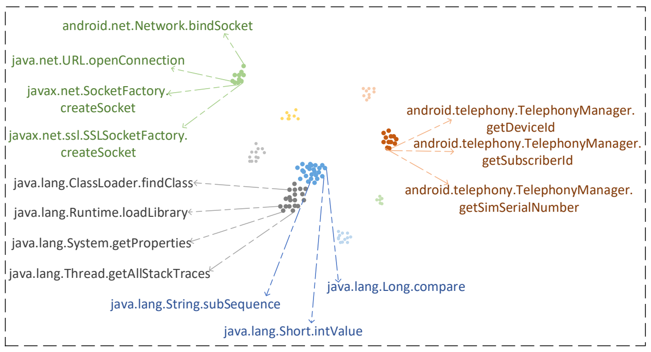


Figure 8: [RQ3] Continuous feature stability scores across ten groups of four malware families.



android.net.Network.bindSocket
java.net.URL.openConnection
javax.net.SocketFactory.createSocket
javax.net.ssl.SSLSocketFactory.createSocket
java.lang.ClassLoader.findClass
java.lang.Runtime.loadLibrary
java.lang.System.getProperties
java.lang.Thread.getAllStackTraces
java.lang.String.subSequence
java.lang.Short.intValue
java.lang.Long.compare
android.telephony.TelephonyManager.getDeviceId
android.telephony.TelephonyManager.getSubscriberId
android.telephony.TelephonyManager.getSimSerialNumber

Figure 9: [RQ4] Visualizing APIs used in the motivating example (§2) and the "java.lang" package.

relations and entities have already demonstrated their capability in grouping semantically-close APIs (see RQ4 in §5.3) and improving existing malware classifiers (see RQ1 in §5.1 and RQ2 in §5.2).

Threshold in Majority Voting of AVs. We follow DroidEvolver in adopting 15 AVs as the threshold to label an app as a malware. Other work [3, 39] may use different thresholds considering their experimental settings. A recent paper [54] justifies that any number between 2 and 39 is reasonable.

Relation Extraction from Code Analysis. Some of the relations extracted from API documents by APIGraph, e.g., "returns" and "throws", are also available in the Android framework code. We choose to design APIGraph to extract relations from API documents, because we believe that it is also how malware developers understand Android APIs and make corresponding changes for evolutions. We leave it as a future work to extract and compare relations from Android source code. Furthermore, documentation mining is not the only way to extract such relations. Other solutions such as API usage mining with large-scale market apps [14, 38] may also generate similar relations.

Non-API-based Malware Classifiers. APIs are a popular type of features widely adopted by many other existing malware classifiers [1, 11, 12, 22], mainly because APIs are essential in implementing malware functionalities. There indeed are two types of classifiers that do not directly adopt APIs as a feature. First, some classifiers, e.g., Mclaughlin et al. [33], adopt opcodes and n-gram as features: Although APIs are not explicitly used as a feature, they are implicitly embedded as part of the opcodes. We believe that APIGraph can still help such classifiers by transforming those opcodes to incorporate API cluster information. Second, some classifiers, e.g., MassVet [10], mainly adopt UI structures for malware detection. Such classifiers may age quickly given malware evolution because those features like UI structures are unreliable and easy to change.

Malware Obfuscation. Android apps may obfuscate themselves via many techniques, such as Java reflection, packing [15], and dynamic code loading [16] to bypass existing analysis. This is an orthogonal problem to what has been studied in APIGraph and one should refer to existing works [6, 26, 40, 53] for solutions.

Merits beyond Android. The idea of APIGraph can also benefit malware detection on Windows and iOS, given their APIs' semantic similarity. In the future, we will consider expanding APIGraph to these tasks.

7 RELATED WORK

Android Malware Classifiers. Machine learning (ML) has been widely used to detect Android malware in both academic and industry environments. One popular, yet wildly adopted features in ML-based Android classifiers is APIs provided by the Android framework: For example, a majority of previous works [1, 3, 11, 12, 22, 32, 50–52] rely on APIs used by the apps as the features to detect malice. Specifically, DroidAPIMiner[1] and DREBIN [3] use the occurrence of APIs; DroidEye [11] and StormDroid [12] use API usage frequency; MalDolzer [22] adopts API calling sequences; and DroidMiner [50], DroidSift [52], and AppContext [51] adopt API call graph.

It is worth noting that most of prior works treat each API separately and ignore the semantic relations among these APIs. MaMaDroid [32] is one of the few exceptions that abstract APIs to corresponding packages, but such a coarse-grained grouping cannot effectively capture API semantic relations either. APIGraph is able to enhance those API-based Android malware classifiers to capture malware evolution, thus slowing down aging.

Concept Drift and Model Aging. Concept drift is a common phenomenon in machine learning, where the statistical properties of the samples change over time. Concept drift causes that machine learning trained models to fail to work on new testing samples, which is known as model aging [49], or time decay [39], or model degradation [24] and deterioration [9] in the literature. Transcend [21] proposes to use statistic techniques to detect concept drift before the model's performance starts to fall sharply. Tesseract [39] proposes a new metric named AUT (Area Under Time) to effectively measure how a model performs over time in the setting of concept drift. It also points out that when training and testing models, spatial and temporal constraints must be satisfied to faithfully reflect the model's performance, and in this paper we exactly follow these constraints to set up our large-scale dataset. MaMaDroid [32] notices that the change in used APIs can affect the performance of the trained models, so it abstracts API calls to their packages and families. However, as shown in this paper the relations between APIs are much broader than the package relation, and capturing more relations between APIs can help MaMaDroid perform better detect evolved malware. EveDroid [24] and DroidSpan [9] try to find more sophisticated and distinguishable features in behavioral patterns and information flow and then build more sustainable models. Unlike these two approaches that rely on their chosen features and underlying algorithms, we propose to let models capture relations between APIs, and our method is more general and can be used to enhance existing malware classifiers.

As a general comparison, APIGraph is the first work that extracts the underlying reasons for model aging: That is, Android malware keeps evolving with similar functionalities but varied implementations. Therefore, APIGraph, being orthogonal to existing ML-based approaches like retraining, active learning, and online learning, can enhance existing ML classifiers to be aware of malware evolution, thus slowing aging.

Semantics from API Documentation. Knowledge graphs [7, 17] have been successfully constructed and applied to in many real-world tasks, such as extracting information and answering questions. Inspired the concept of knowledge graph, we propose API relation graph to represent the internal relations among diverse Android programming entities. The major challenges here are that we need to extract and represent Android specific entities and relations. Several knowledge graph embedding algorithms have been proposed, including TransE [8], TransH [47], and TransR [28]. Our API embedding algorithm uses the TransE with some variations to convert APIs in the relation graph to embeddings.

The Android API reference documents contain abundant information about APIs. Maalej et al. [30] have developed a taxonomy of knowledge types in API reference documents. Based on this taxonomy, Li et al. [25] use NLP techniques and define templates to extract API caveats (i.e. facts that developers should know to avoid unintended use of APIs) from API documents. As a comparison, the purpose of APIGraph is to extract semantic similarity among APIs so that such similarities can capture the preserved semantics during malware evolution.

8 CONCLUSION

Android malware keeps evolving over time to avoid being detected by existing classifiers. This paper proposes APIGraph to capture semantic similarity among APIs, called API semantics, and enhance state-of-the-art classifiers with API semantics so that they can still classify evolved malware samples. Specifically, APIGraph builds a so-called API relation graph, coverts each API entity in the graph to an embedding, called API embedding, and then groups APIs in the embedding form into clusters. Those clusters are used to replace each individual API used by state-of-the-art classifiers as a feature. We applied APIGraph to enhance four state-of-the-art classifiers and evaluated them using a dataset created by ourselves which contains more than 322K Android apps ranging from 2012 to 2018. Our evaluation shows that APIGraph can significantly reduce the number of samples to label in those four classifiers by 32%–96%. To facilitate any follow-up research, we have publicly released our dataset and source code at https://github.com/seclab-fudan/APIGraph.

ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their helpful comments. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1836210, U1836213, U1736208, 61972099), Natural Science Foundation of Shanghai (19ZR1404800), and National Program on Key Basic Research (NO. 2015CB358800). The authors from Johns Hopkins University were supported in part by National Science Foundation (NSF) grants CNS-18-54000. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

REFERENCES

[1] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. DroidAPIMiner: Mining API-level Features for Robust Malware Detection in Android. In *Proceedings of the International Conference on Security and Privacy in Communication Systems (SecureComm)*. Springer, 86–103.

[2] Apktool. 2019. A Tool for Reverse Engineering Android APK Files. https://ibotpeaches.github.io/Apktool/.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 23–26.

[4] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*. ACM, 217–228.

[5] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-visiting Android Permission Specification Analysis. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*. 1101–1118.

[6] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents. In *Proceeding of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 669–679.

[7] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1247–1250.

[8] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Proceedings of the 26th Advances in Neural Information Processing Systems (NIPS)*. 2787–2795.

[9] Haipeng Cai. 2020. Assessing and Improving Malware Detection Sustainability through App Evolution Studies. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 28.

[10] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 seconds: Mass Vetting for New Threats at the Google-play Scale. In *Proceedings of 24th USENIX Security Symposium (USENIX Security)*. 659–674.

[11] Lingwei Chen, Shifu Hou, Yanfang Ye, and Shouhuai Xu. 2018. DroidEye: Fortifying Security of Learning-based Classifier against Adversarial Android Malware Attacks. In *Proceedings of 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 782–789.

[12] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. 2016. StormDroid: A Streaminglized Machine Learning-based System for Detecting Android Malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 377–388.

[13] Universit d du Luxembourg. 2016. AndroZoo. https://androzoo.uni.lu/.

[14] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 472–489.

[15] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. 2018. Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[16] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. 2015. Grab'n Run: Secure and Practical Dynamic Code Loading for Android Applications. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*. 201–210.

[17] Google. 2020. Google - Introducing the Knowledge Graph: Things, Not Strings. https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html.

[18] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial Examples for Malware Detection. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 62–79.

[19] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. 2016. Deep4MalDroid: A Deep Learning Framework for Android Malware Detection based on Linux Kernel System Call Graphs. In *Proceedings of 2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*. IEEE, 104–111.

[20] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. 2017. Euphony: Harmonious Unification of Cacophonous Anti-virus Vendor Labels for Android Malware. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. IEEE Press, 425–435.

[21] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting Concept Drift in Malware Classification Models. In *Proceedings of 26th USENIX Security Symposium (USENIX Security)*. 625–642.

[22] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. 2018. MalDozer: Automatic Framework for Android Malware Detection Using Deep Learning. *Digital Investigation* 24 (2018), S48–S59.

[23] Kaspersky. 2019. Machine Learning Methods for Malware Detection. https://media.kaspersky.com/en/enterprise-security/Kaspersky-Lab-Whitepaper-Machine-Learning.pdf.

[24] Tao Lei, Zhan Qin, Zhibo Wang, Qi Li, and Dengpan Ye. 2019. EveDroid: Event-Aware Android Malware Detection Against Model Degrading for IoT Devices. *IEEE Internet of Things Journal (IOTJ)* (2019).

[25] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving API Caveats Accessibility by Mining API Caveats Knowledge Graph. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193.

[26] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: Taming Reflection to Support Whole-program Analysis of Android Apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. 318–329.

[27] Hu X Li Y, Jang J. 2019. AMD Dataset. http://amd.arguslab.org/sharing.

[28] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. 2015. Learning Entity and Relation Embeddings for Knowledge Graph Completion. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI)*.

[29] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. 2015. Marvin: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis. In *Proceedings of IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. IEEE, 422–433.

[30] Walid Maalej and Martin P Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering (TSE)* 39, 9 (2013), 1264–1282.

[31] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data Using t-SNE. *Journal of machine Learning Research* 9, Nov (2008), 2579–2605.

[32] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[33] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. 2017. Deep Android Malware Detection. In *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 301–308.

[34] Trend Micro. 2018. The Evolution of XLoader and FakeSpy: Two Interconnected Android Malware Families. https://documents.trendmicro.com/assets/pdf/wp-evolution-of-xloader-and-fakespy-two-interconnected-android-malware-families.pdf.

[35] Trend Micro. 2018. XLoader Android Spyware and Banking Trojan Distributed via DNS Spoofing. https://blog.trendmicro.com/trendlabs-security-intelligence/xloader-android-spyware-and-banking-trojan-distributed-via-dns-spoofing/.

[36] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*. 3111–3119.

[37] Annamalai Narayanan, Liu Yang, Lihui Chen, and Liu Jinliang. 2016. Adaptive and Scalable Android Malware Detection through Online Learning. In *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2484–2491.

[38] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API Embedding for API Usages and Applications. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 438–449.

[39] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. USENIX Association, Santa Clara, CA, 729–746.

[40] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications.. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Vol. 14. 23–26.

[41] scikit-learn. 2020. scikit-learn, Machine Learning in Python. https://scikit-learn.org.

[42] spaCy. 2020. spaCy - Industrial-Strength Natural Language Processing. https://spacy.io/.

[43] MA Syakur, BK Khotimah, EMS Rochman, and BD Satoto. 2018. Integration K-means Clustering Method and Elbow Method for Identification of the Best Customer Profile Cluster. In *IOP Conference Series: Materials Science and Engineering*, Vol. 336. IOP Publishing, 012017.

[44] TensorFlow. 2020. TensorFlow - An End-to-end Open Source Machine Learning Platform. https://www.tensorflow.org/.

[45] VirusShare. 2020. VirusShare. https://virusshare.com.

[46] VirusTotal. 2020. VirusTotal. https://virustotal.com.

[47] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge Graph Embedding by Translating on Hyperplanes. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI)*.

[48] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, Bonn, Germany, 252–276.

[49] Ke Xu, Yingjiu Li, Robert Deng, Kai Chen, and Jiayun Xu. 2019. Droidevolver: Self-evolving android malware detection system. In *2019 IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE, 47–62.

[50] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. 2014. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 163–182.

[51] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE Press, 303–313.

[52] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 1105–1116.

[53] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 293–311.

[54] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. 2020. Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines. In *Proceedings of the29th USENIX Security Symposium (USENIX Security)*.

A MALICIOUS FAMILY

Table 10 lists the top 30 malware families adopted in our evaluation.

Table 10: Top 30 Malware Families with The Most Number of Samples in Our Dataset.

Top	Family	# APKs	Start	End
1	smsreg	21,730	2012-02	2018-12
2	airpush	9,434	2012-01	2018-07
3	dowgin	5,614	2012-06	2018-11
4	smspay	4,074	2012-02	2018-12
5	boxer	3,021	2012-01	2018-06
6	triada	2,777	2015-01	2018-11
7	youmi	2,294	2012-02	2018-11
8	kuguo	2,227	2012-02	2018-11
9	fakeinst	1,751	2012-01	2018-07
10	adwo	1,742	2012-01	2018-11
11	mecor	1,638	2013-09	2016-03
12	skymobi	1,537	2013-05	2018-09
13	opfake	1,416	2012-01	2018-08
14	fusob	1,303	2015-04	2018-07
15	waps	1,122	2012-01	2018-12
16	dnotua	1,071	2012-09	2018-12
17	secapk	1,048	2013-09	2018-11
18	artemis	977	2012-01	2018-12
19	wapsx	963	2012-02	2018-08
20	hiddenads	723	2014-08	2018-07
21	hiddad	698	2015-11	2018-11
22	hiddenapp	625	2014-07	2018-07
23	plankton	584	2012-01	2018-07
24	admogo	580	2012-02	2018-07
25	smforw	555	2013-05	2018-11
26	utchi	553	2012-11	2018-07
27	smsspy	541	2012-03	2018-11
28	appquanta	536	2013-10	2018-06
29	lotoor	520	2012-01	2018-06
30	umpay	482	2012-09	2018-12