Slowing Down the Aging of Learning-based Malware Detectors with API Knowledge

Xiaohan Zhang, Mi Zhang, Yuan Zhang, Ming Zhong, Xin Zhang, Yinzhi Cao, and Min Yang

Abstract—Learning-based malware detectors are widely used in practice to safeguard real-world computers. One major challenge is known as model aging, where the effectiveness of these models drops drastically as malware variants keep evolving. To tackle model aging, most existing works choose to label new samples to retrain the aged models. However, such data-perspective methods often require excessive costs in labeling and retraining. In this paper, we observe that during evolution, malware samples often preserve similar malicious semantics while switching to new implementations with semantically equivalent APIs. Such observation enables us to look into the problem from a different perspective: feature space. More specifically, if the models can capture the intrinsic semantics of malware variants from feature space, it will help slow down the aging of learning-based detectors. Based on this insight, we design APIGRAPH to automatically extract API knowledge from API documentation and incorporate these knowledge into the training of malware detection models. We use APIGRAPH to enhance 5 state-of-the-art malware detectors, covering both Android and Windows platforms and various learning algorithms. Experiments on large-scale, evolutionary datasets with nearly 340K samples show that APIGRAPH can help slow down the aging of these models by 5.9% to 19.6%, as well as reduce labeling efforts from 33.07% to 96.30% on top of data-perspective methods.

Index Terms—Malware Detection, Model Aging, API Knowledge, Learning-based Detection.

1 INTRODUCTION

LEARNING-based malware detectors [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15] are becoming more and more popular in both academia and industry, because they do not rely on explicitly defined signatures nor rules, thus being more scalable and flexible than signature-based or rule-based malware detectors. However, as malware samples keep evolving (e.g., adding more functionalities or deliberately evading detection [16]), the effectiveness of learning-based malware detectors drops significantly, which is known as *model aging*, or time decay [16], [17], model degradation [18], and deterioration [19] in the literature. In 2019, Kaspersky points out in a white paper [20] that the detection rate of one of its commercial learning-based detectors drops from almost 100% to below 60% in only three months. Model aging has become one major obstacle to the practicalness of learning-based malware detectors.

To tackle the aging of learning-based malware detectors, existing solutions adopt data-perspective approaches, which retrain and update aged models [10], [12], [21] with newly labeled samples. However, data-perspective methods have two shortcomings: 1) Labeling samples and retraining models come at huge costs, as they heavily require expert knowledge and computing resources. Although optimizing strategies such as incremental/online learning [10] and active learning [16] have been proposed, they still require large amounts of newly-labeled samples (as verified in our experiments in §6.2). 2) The retrained models are still unaware of malware evolution, thus they need frequent retraining and updating [22], [23]. As a result, new methods are needed to help mitigate the problem of model aging in malware detection.

In this paper, we look into the model aging problem from a new *feature space* perspective. Our key observation is that malware samples, during evolution, often keep the same or similar semantics but switch to a different implementation using semantically equivalent APIs. For example, the original malware may send one user identifier like IMEI via HTTP requests, but its evolved variant could send a different identifier such as IMSI via sockets. Semantically, they are almost the same, but the directly observed implementations are different. Therefore, if such semantic similarity can be captured and incorporated into machine learning models, it can help slow down the aging of these detectors.

Based on the above idea, we propose APIGRAPH, a system that can extract semantic knowledge from API documents, called *API knowledge*, and incorporate these knowledge into existing malware detectors to slow down their aging. First, APIGRAPH leverages natural language processing (NLP) techniques and predefined templates to extract API entities and relations from official documents, and builds an *API relation graph*. The API relation graph can faithfully reflect the semantic relations among different APIs. After that, APIGRAPH extracts API knowledge from the relation graph by converting each API entity into an embedding representation and grouping semantically-close APIs into the same clusters. The extracted API knowledge in the format of API clusters can be used in exiting malware detectors to help them capture the intrinsic semantics during malware evolution.

To evaluate its effectiveness, we use APIGRAPH to enhance 4 state-of-the-art Android malware detectors [3], [9], [11], [12] and 1 Windows malware detector [24]. For each platform, we build a large-scale, evolutionary dataset satisfying both temporal and spatial consistency, as pointed out by a recent best practice [16], to fairly evaluate model aging. Specifically, the Android dataset contains more than 322K Android apps ranging from 2012 to 2018, which is almost three times the one used in the SOTA work [16], and the Windows dataset contains about 17K samples which is the first evolutionary dataset following the guidelines in [16].

We conduct extensive experiments to evaluate the effectiveness of APIGRAPH in enhancing these baselines, including: 1) prolonging model lifetime, 2) reducing maintaining efforts, 3) stabilizing feature space, 4) capturing API closeness, and 5) robustness against adversarial attacks. The results show that APIGRAPH can effectively slow down the aging of these malware detectors. First, APIGRAPH can prolong models' lifetime by 19.2%, 19.6%, 15.6%, 8.7% respectively for the 4 Android malware detectors respectively, and 5.9% for the Windows malware detector. Also, APIGRAPH can reduce maintaining efforts even on top of the most optimized data-perspective method [16]: the number of samples needed to be labeled can be reduced by 33.07%∼96.30%, and the retrain frequency is also significantly decreased. The visualized results can vividly reflect how APIGRAPH can help stabilize feature space and capture API closeness. Finally, APIGRAPH can help improve the robustness of existing malware detectors against adversarial attacks.

Contributions. This paper makes the following contributions.

- We study how to slow down model aging of ML-based malware detectors from a new perspective other than existing data-driven methods — API feature space. Our observation is that during evolution, malware samples tend to preserve similar malicious functionalities but use different API implementations. Therefore, we propose to incorporate API knowledge into ML models to help them capture the intrinsic semantics among malware variants.
- Based on the above idea, we design and implement APIGRAPH that can help extract API knowledge from the API documentation. It builds an API relation graph using NLP techniques and pre-defined templates, and then uses API embedding and API clustering to group semantically-close APIs into clusters to enhance existing malware detectors.
- We evaluate APIGRAPH on 4 Android malware detectors and 1 Windows malware detector, covering both traditional ML and deep learning models. Experiments on large-scale, evolutionary datasets with nearly 340K samples (the largest of its kind, as far as we know) show that APIGRAPH can help slow down the aging of existing malware detectors by 5.9% to 19.6%. Compared to existing data-perspective methods, it can also significantly save maintaining efforts. Finally, we release the source code and datasets at https://github.com/seclab-fudan/APIGraph to facilitate subsequent studies.

Organization. § 2 uses a motivating example to explain how APIGRAPH slows down the aging of malware detectors from the feature space and gives an overview of the system architecture. § 3 describes the design of APIGRAPH in building and leveraging API relation graphs. § 4 reports the API relations graphs built by APIGRAPH for both Android and Windows platforms. § 5 introduces our experimental setup and § 6 reports the evaluation results of APIGRAPH in enhancing five baselines. § 7 discusses some limitations of APIGRAPH and § 8 discusses the most related work. At last, § 9 concludes the paper.

2 OVERVIEW

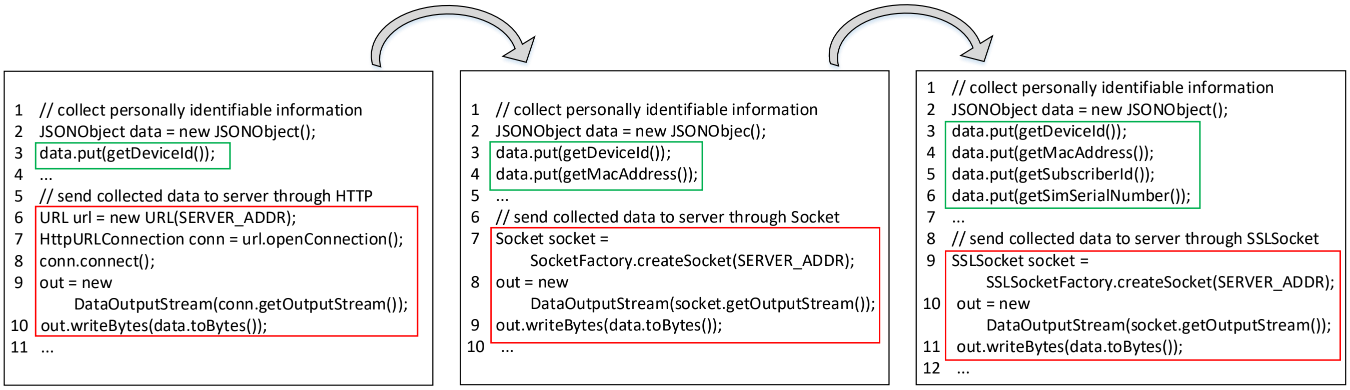
In this section, we start from a motivating example and then give an overview of the system architecture.

2.1 A Motivating Example

According to previous studies [25], removing, replacing, and adding API calls to the code while not affecting their original malicious functionalities, are common tricks used during malware evolution. In this paper, we first use a real-world malware, called `XLoader`, to illustrate how APIGRAPH captures the semantics across various malware versions during evolution. According to the reports by TrendMicro [26], `XLoader` is a spyware and banking trojan that steals personally identifiable information (PII) and financial data. We observe that although `XLoader` has evolved into six different variations with significant implementation changes from April 2018 until late 2019, many semantics across these variations remain the same.

To ease the illustration of this observation, we reverse three `XLoader` variations and simplify their implementations in Figure 1. From this figure, we can find two types of semantics that are preserved across the 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 sources in V2 and then to multiple sources in V3. Specifically, V1 only collects the device ID, i.e., the IMEI; V2 adds the MAC address; and V3 adds IMSI and ICCID. Second, the malware sends PII to the malware server via three different channels, 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).

We then explain how APIGRAPH captures the semantic similarity among the three versions of `XLoader` in terms of sending PII and thus helps ML detectors trained with V1 samples to detect evolved V2 and V3 samples. Figure 2 shows a small part of the relation graph constructed by APIGRAPH (§ 3.1 & 3.2), which captures the relations among some Android APIs, permissions, and exceptions. All the mentioned APIs in Figure 1 (i.e., `openConnection`, `SocketFactory.createSocket`, and `ssl.SSLSocketFactory.createSocket`) throw `IOException` and use `INTERNET` permission. Besides, some of them have more similar behaviors in throwing exceptions and using permissions. That is, the three APIs are close enough in terms of their neighborhoods in the graph, and can be grouped together in a cluster. Therefore, an ML detector, with the help of the relation graph, can capture the similarity between V2/V3 and V1 and detect V2 and V3 as malware after the evolution. For example, several Android malware detectors [3], [9], [12] use an API occurrence vector to represent each app. Therefore, the feature vectors generated by these detectors from V2/V3 will be significantly different from those for V1. With



Listing 1: pseudo-code of XLoader V1 Listing 2: pseudo-code of XLoader V2 Listing 3: pseudo-code of XLoader V3

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

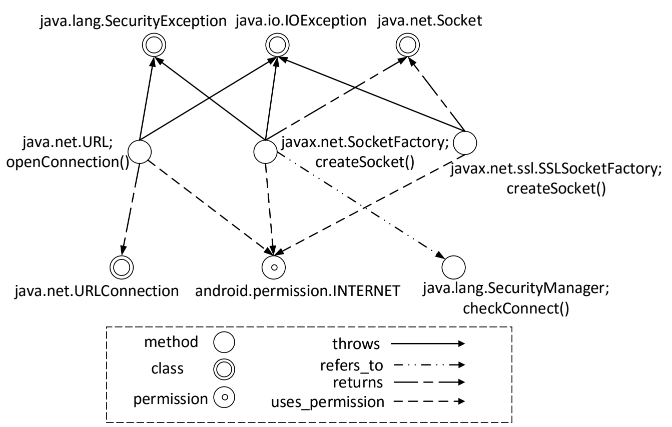


Fig. 2. Part of the API relation graph generated by APIGRAPH, showing the semantic closeness of different APIs used by XLoader variations in Figure 1.

APIGRAPH, existing detectors can be enhanced to use the clusters to represent APIs. In this way, cluster occurrence vectors generated from V1 and V3 samples will be quite similar; therefore the detectors can detect the evolved V2/V3 samples even when trained with only V1 samples.

2.2 System Architecture

Figure 3 shows the overall architecture of APIGRAPH. A key concept introduced by APIGRAPH is the API relation graph, which is used to capture the semantic relations among different APIs. There are two major phases in APIGRAPH: building API relation graph and leveraging API relation graph. First, APIGRAPH builds an API relation graph by collecting API documents and extracting entities–such as APIs and permissions–and relations between those entities, with the help of NLP techniques. 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 vectors of two entities in the embedding space reflect the semantic closeness of the relation between them. Therefore, APIGRAPH generates the entity embedding as solving an optimization problem to make the vectors of two entities with the same relation as close as possible. Based on the embedding vectors of APIs, APIGRAPH groups similar APIs into clusters. These API clusters are further used to enhance existing detectors so that they can capture the semantically equivalent evolution among malware samples.

3 APIGRAPH DESIGN

In this section, we first define the concept of the API relation graph and then describe how to build and leverage the API relation graph to slow down model aging.

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, which means that entities and relations have different types. Previous work [27] has developed a taxonomy of entities and relations in API documents. In this paper, beyond this taxonomy, we also consider other entities and relations under the context of malware detection. For example, the permissions in Android are considered because they are essential to malware analysis. To generalize the entities and relations in an API relation graph for Android and Windows, we group different entities and relations into different categories.

Entity Types. Entities are basic elements in an API relation graph. We mainly consider three categories of entities:

- **Functional Unit** is marked as u, which is the fundamental unit defined in the API documents for developers to use. For the Android platform, the fundamental unit is a *method*, while for Windows it can be a *function* (including macros and methods), or a *struct*.
- **Container** is marked as c, which is used by the platform to organize the functional units. Besides, containers are hierarchical, which means a high-level container may be composed of multiple low-level containers. For Android, a container may be a *class* and a *package*, and a *package* may have multiple *class*es; For Windows, a container may be a *class* (including interfaces), or a *header* file.
- **Permission** is marked as p, and is used to specify the capability that a functional unit or a container

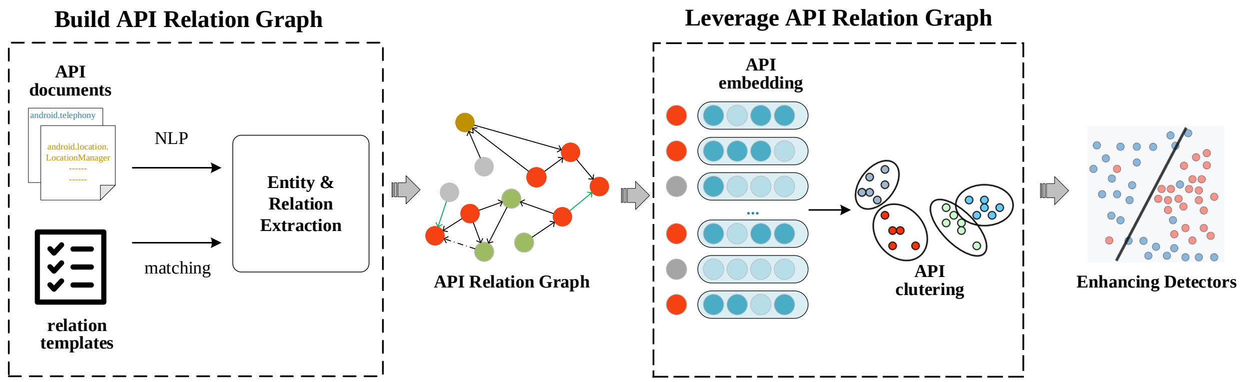


Fig. 3. The overall architecture of APIGRAPH.

requires. For Android, some methods can be called only when the program has the corresponding *permissions*.

Relation Types. Relations capture how entities are related to one another. We consider three categories of relations among the above entities:

- **Structure** category describes the organization relations between two functional units, a functional unit, and a container, and two containers. For example, in Android, a *function_of* relation connects a *method* to its belonging *class*, while *inheritance* relation connects a *class* entity with its inherited *class* entity. Also, *uses_parameter*, *returns*, *throws* relations reflect one *method* entity may use a *class* entity as its parameter, return value, or thrown exception respectively.
- **Reference** category describes the relations between two functional units, and a functional unit and its container. There are three relation types in this category, including *conditional*, *alternative*, and *refers_to* relations for both Android and Windows. For example, a *conditional* relation specifies that one *method* entity conditionally depends on another *method* entity, e.g., one API should be used only after another API has been called. An *alternative* relation depicts that one *method* entity can be replaced by another *method* entity. In addition, a *refers_to* relation describes a general relationship between two entities. For example, the API document may refer to another *method* entity when describing one *method* entity using a sentence like "see also ...".
- **Permission** category contains the *uses_permission* relation to describe that a *method* entity may require a *permission* entity.

The full list of the relations and the entities they connect are listed in Table 1.

3.2 Building API Relation Graph

To build API relation graphs, APIGRAPH first collects API documents and then parses these documents to extract entities and their relations, with the help of NLP techniques. After that, entities are linked with their relations to form a heterogeneous directed graph. Next, we use the Android platform to illustrate the processes of building API relations graphs from API documents and highlight our special handling to Windows when needed.

API Documents Collection. APIGRAPH downloads the Android API reference documents for all platform APIs and support libraries from the official website [28]. In the Android platform, different Android versions have corresponding API levels, e.g. the API level for Android 10 is 29. Since the major active Android versions at present are Android 4.0-10 [29], APIGRAPH collects the API documents for their corresponding Android levels, i.e., API level 14-29. For the Windows platform, APIGRAPH downloads the API documents for Windows 10 from the official website [30]. These documents are collected as HTML files and are further parsed into JSON files to ease subsequent processing.

Entity Extraction. API documents are organized in hierarchies. For example, the packages, classes, and methods for the Android API documents can be accessed from the top level to the bottom level. By parsing the organization tree, APIGRAPH can extract all the entities from the *functional unit* and *container* categories. Furthermore, all the *permission* entities are extracted from the manifest file [31]. The entity extraction process on the Windows platform is similar to this step.

Relation Extraction. Relations under different categories are extracted in different ways. For relations under the *structure* category, *function_of*, *class_of*, and *inheritance* relations are extracted from the hierarchical organization of *function units* and *containers*; *inheritance* relations are extracted from the class definitions; *uses_parameter*, *returns*, and *throws* relations are extracted directly from the method prototypes.

For relations that belong to *reference* and *permission* categories, they can only be extracted from the text descriptions of each functional unit. We use Figure 4 as an example to illustrate this process. In Figure 4, three paragraphs are describing the functional unit entity `getDeviceId`. The first paragraph, P1, states that this method is deprecated in higher API levels (26 and above), and two methods `getImei` and `getMeid` are recommended for replacement. In this case, APIGRAPH extracts two *alternative* relations between `getDeviceId` and `getImei`, and between `getDeviceId` and `getMeid`. From the last paragraph, P3, APIGRAPH extracts two *uses_permission* relations between `getDeviceId` and `READ_PRIVILEGED_PHONE_STATE`, `getDeviceId` and `READ_PHONE_STATE`, and one *refers_to* relation between `getDeviceId` and `hasCarrierPrivileges`.

TABLE 1
Relation types defined in APIGRAPH, where "u", "c", "p" represents the functional unit, container, and permission respectively.

Category	Relations	Connected Entities: Android	Connected Entities: Windows
Structure ($u \to c$, $c \to c$, $u \to u$)	function_of	method → class	function → class, function → header
	class_of	class → package	class → header
	inheritance	class → class	class → class
	uses_parameter	method → class	function → function, function → struct, function → class
	throws	method → class	\
	returns	method → class	function → struct
Reference ($u \to u$, $u \to c$)	conditional	method → method	function → function
	alternative	method → method	function → function
	refers_to	method → method, method → class	function → function, function → struct, function → class
Permission ($u \to p$)	uses_permission	method → permission	\

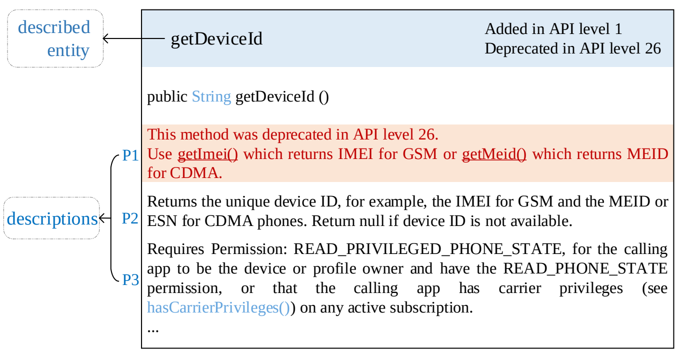


Fig. 4. The description for *android.telephony.TelephonyManager. getDeviceId()*.

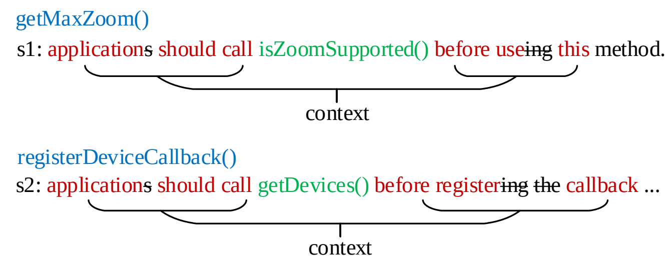


Fig. 5. s1 and s2 are descriptive sentences for two described entities: *getMaxZoom* and *registerDeviceCallback* respectively. The describing entities (*isZoomSupported* and *getDevices*) are marked in green while their contexts are in red ($w = 6$). Note that "using" and "registering" are reduced to their base forms and "the" is removed during normalization.

However, it is impractical to manually extract these relations from unstructured texts one by one, as there are a large number of APIs (e.g., Android API level 29 has about 50K APIs and the latest Win32 API list has about 40K APIs). Our key observation is that there are some common patterns in describing the relations between entities. Therefore, we can summarize these patterns with templates and use these templates to extract relations. Specifically, APIGRAPH leverages NLP techniques and designs a template-based matching method to extract *reference* and *permission* relations from the unstructured text descriptions. In our previous work [32], we use an iterative workflow to manually summarize templates from unstructured descriptions, which requires huge manual efforts. In this paper, we improve such practice by introducing a clustering-based template generation method, which first groups similar short sentences into a cluster, and then summarizes templates from these clusters. As a result, we can not only save manual efforts but also accelerate the template generation process.

3.2.1 Clustering-based Template Generation

To ease the illustration, we first define the following terms:

- *Described entity* is the target entity of each description.
- *Describing entity* is the entities mentioned in the description that may have some relations with the described entity.

For example, in Figure 4, `getDeviceId` is the described entity, while `getImei`, `getMeid`, `READ_PHONE_STATE`, `READ_PRIVILEGED_PHONE_STATE` are all describing entities. Note that there may be no or multiple describing entitie(s) in the description for each described entity.

Our template generation process is context-sensitive, which consists of three steps: first, APIGRAPH uses NLP tools to split every description into sentences and normalize each sentence; second, it extracts a short context for each describing entity and clusters these describing contexts into different groups based on context similarity; at last, relation templates are summarized from these context groups, and then used to extract relations automatically.

1) *Sentence splitting and normalization*. APIGRAPH splits the descriptions into sentences and uses *lemmatization* to transform each word to its base form (e.g., both "requires" and "required" are transformed to "require"). Besides, meaningless words such as definite and indefinite articles are removed from the sentence. Further, the names of describing entities are unified to ensure that each entity has only one unique name when polymorphic names exist. For example, the name *android.Manifest.permission.INTERNET* and its constant value "android.permission.INTERNET" are both used in documents, but they refer to the same entity. To unify this entity, APIGRAPH replaces the former one with the latter one.

2) *Context extraction and clustering*. From the normalized sentence, we use the words around the describing entity as its *describing context*. As shown in Figure 5, a window w is used to control the number of words in the describing context. These contexts are then clustered into different

Algorithm 1 API Embedding and Clustering

Input: Relation graph $G = \langle E, R \rangle$, learning rate λ, embedding size k, cluster size C.

Set triples $S = \emptyset$ $\triangleright$ Form Training Set
Add existing relations to triples S
for each entity $e \in E$ **do** $\triangleright$ Vector Initialization
    Assign e with a vector $l_e \in \mathbb{R}^k$
for each relation $r \in R$ **do**
    Assign r with a vector $l_r \in \mathbb{R}^k$
while True **do** $\triangleright$ Train Embeddings
    for triple $(h, r, t) \in S$ **do**
        Minimize the following loss function:

$$\ell = \|l_h + l_r - l_t\|_2^2$$

        Update l_h by gradient descent:

$$l_h = l_h + \lambda \cdot \frac{\partial \ell}{\partial l_h}$$

        Update $l_r, l_t, l_{t'}$ with gradient descent similarly
    if embeddings do not change **then**
        break
Collect embeddings of method entities $\triangleright$ Cluster APIs
Use k-Means algorithm to find C clusters

groups according to their similarity. The similarity between two contexts is calculated using Jaccard similarity. For example, in Figure 5, the two describing contexts in s1 and s2 have 8 unique words in all (words in red color), and share 4 words (i.e. "application", "should", "call", "before"). Therefore, their context similarity is 0.5 (4/8). When the similarity score of two contexts is not less than a threshold, they are clustered into the same group. According to our experience, this threshold is set to 0.5.

3) *Template generation*. Based on the clustered describing contexts, we manually summarize relation templates from them. To ease template matching, regular expressions are used to represent the relation templates. For example, a template, "should call ENT before" is summarized from Figure 5 which describes a *conditional* relation. More examples of the summarized relation templates are given in Table 2.

3.3 Leveraging API Relation Graph

Considering that different ML algorithms may require different input formats (as shown in Table 6), we need a general and easy-to-use way to incorporate API knowledge into these models. Our idea is to group semantically close APIs into a cluster and use the clusters to represent those APIs. By abstracting APIs to clusters, the underlying models can better capture the malicious behaviors inside the evolved malware samples, even these samples and variants have different implementations and use different APIs.

To get the API clusters from the API relation graph, we propose a two-step method: 1) *API embedding*, which encodes each API in the API relation graph into a vector; and 2) *API clustering*, which groups semantically close APIs into different clusters based on their embedding vectors. The pseudo-code is illustrated in Algorithm 1, and we introduce the most important details below.

API Embedding. The idea of API embedding is inspired by word embedding [33] and graph embedding [34], [35], [36]. It aims to convert each API in the graph into a vector so that the APIs that are semantically closer have a higher similarity between their vectors. To this end, we leverage a prior algorithm called TransE [34] and fit TransE into our API knowledge problem, as described in Algorithm 1. Specifically, suppose we have a relation r that links an entity h to an entity t and three vectors l_h, l_r, l_t are used to represent them, the core idea of TransE is to continuously adjust the three vectors so that the sum of l_h and l_r is as close as to l_t. In this way, semantically close APIs will have similar vector representations.

API Clustering. After APIs are represented using vectors, we can then group semantically close APIs into the same cluster. Our idea is to use the k-Means algorithm, which partitions the APIs into k clusters and minimizes the within-cluster sum of squares. We rely on Elbow [37] method to determine the final cluster number.

4 Statistics of API Relation Graph

We implement a prototype of APIGraph, which contains 3,344 lines of Python code, including API documents collecting and parsing, entity and relation extracting, relation graph building, and API embedding and clustering. Some modules are built on existing libraries. For example, we use spaCy [38] (a Python NLP toolkit) for text processing, TensorFlow [39] for API embedding, and sklearn [40] for API clustering.

Extracted Entities and Relations. We report the generated API relation graphs from their entities and relations. Table 3 shows the number of extracted entities for Android API level 29 and Windows 10. Note both Android and Windows have several versions and we use their latest stable versions (at the time of evaluation) as examples. In total, 67,209 and 52,554 entities are extracted for Android and Windows respectively. Table 4 lists the numbers of extracted relations. Note that since *uses_permission* relations in the Android API documents may be incomplete, we also use two API-permission mappings generated by existing works [41], [42] to complement the relations extracted from API documents. In total, 121,345 relations are extracted for Android and 144,594 relations are extracted for the Windows platform.

With these entities and relations, we build API relation graphs and group APIs into different clusters. Following the Elbow method, we choose 2,000 and 1,000 as the cluster number for Android and Windows respectively.

Clustering-based Template Generation. The clustering-based template generation method used in this paper is more efficient than the iterative workflow in our previous work [32]. For example, in our previous work, we need to look into about 150K sentences to summarize the templates for Android 10, which cost three days for two security experts. In contrast, in this work, we only need to investigate about 7K clusters for Android 10, which greatly reduces the manual efforts and speeds up the template generation process. As shown in Table 2, our method generates 217 and 40 templates for Android and Windows platforms respectively.

TABLE 2
Templates that have been summarized from both Android and Windows platforms ('ENT" represents an entity).

Relation Type	Example Templates	# of Templates	
		Android	Windows
conditional	"call ENT before ENT be call", "before ENT return", "if ENT fail", "wait for ENT"	186	29
alternative	"replace by ENT", "use ENT instead"	22	10
refers_to	"see also ENT", "query ENT", "refer to ENT"	5	1
uses_permission	"require permission ENT", "be grant ENT permission"	4	\

TABLE 3
Extracted entities for Android API level 29 and Windows.

Category	Android		Windows	
	Entity Type	Count	Entity Type	Count
Functional Unit	method	59,125	function struct	40,358 6,897
Container	class package	7,368 446	class header	4,363 936
Permission	permission	270	\	\

TABLE 4
Extracted relations for Android API level 29 and Windows platform.

Category	Relation Type	Count	
		Android	Windows
Structure	function_of	59,125	40,531
	class_of	7,368	4,464
	inheritance	3,755	3,482
	uses_parameter	14,528	16,290
	throws	8,310	\
	returns	5,113	454
Reference	conditional	5,990	3,065
	alternative	1,264	80
	refers_to	10,859	76,228
Permission	uses_permission	5,033	\

5 Experimental Setup

In this section, we describe the datasets and existing malware detectors used in our experiments.

5.1 Dataset

Dataset Properties. To evaluate how APIGraph can help to slow down the aging of existing Android/Windows malware detectors, we need first to set up a dataset that is *evolutionary* and *large-scale* for both platforms. Moreover, to make the evaluation fair and reliable, the built dataset should satisfy *temporal consistency* and *spatial consistency* according to the guidelines set by previous works [16]. Specifically, temporal consistency ensures that training samples should be strictly temporally precedent to testing ones, and all testing samples must come from the same period during each testing to eliminate time bias; while spatial consistency ensures that the ratio of malware is close to the percentage of malware in the real-world.

Following the above guidelines, we set up a *large-scale* dataset that contains 322,594 Android samples and 16,953 Windows samples, as shown in Table 5. The number of Android samples is almost two times larger than the one used in previous state-of-the-art work [16]. These samples are from the year 2012 to 2018, which meets the *evolutionary* requirement. Also, we leverage VirusTotal [43] to get the exact appearing time for each sample and make sure that *temporal consistency* is satisfied at the month level during the testing. Referring to previous work [16], we also make sure that malware percentage is close to 10% in each month to meet *spatial consistency*. For Windows, we make sure each test in the evaluation should satisfy this constraint by down-sampling and averaging multiple tests. To get reliable labels for these samples, we rely on VirusTotal to determine whether a sample is benign or malicious. Specifically, following a previous work [12], samples are labeled as malware when at least 15 anti-viruses (AV) engines report them as malicious, while samples are labeled as benign when no AV reports them as malicious. Note that according to a recent study [44] on measuring the labeling effectiveness of malware samples, this strategy is reasonable and stable.

TABLE 5
Evaluation datasets that contain 322,594 Android samples and 16,953 Windows samples across 7 years.

Year	Android		Windows	
	Malicious	Benign	Malicious	Benign
2012	3,066	27,613	239	213
2013	4,871	43,873	2,744	1,871
2014	5,871	52,843	3,330	435
2015	5,797	52,173	2,515	856
2016	5,651	50,859	311	2,880
2017	2,620	24,930	1,019	247
2018	4,213	38,214	192	101
Sum	**32,089**	**290,505**	**10,350**	**6,603**

To ensure the reproducibility of the dataset, all the samples are collected from publicly available repositories. In particular, the Android samples are randomly selected from AndroZoo [45], VirusShare [46], VirusTotal [43], and the AMD dataset [47], [48]; the Windows samples are selected from Ceschin et al. [24]. All these samples have been released to facilitate subsequent researches.

5.2 Evaluated Malware Detectors

As shown in Table 6, we evaluate five state-of-the-art, representative malware detectors that cover both Android and Windows platforms. Specifically, different API feature formats such as occurrence, frequency, or API calls, and different algorithms such as linear algorithms, random forest, and deep neural networks are used in these models. This setting helps to verify the generalization of APIGraph in enhancing state-of-the-art malware detectors.

Reproduction Details. The source code of MaMaDroid [49] and DroidEvolver [50] are publicly available, and we directly use their source code. For the other three detectors whose source code are not available,

TABLE 6
5 Evaluated malware detectors. Note that DroidEvolver uses a model pool that contains 5 linear online learning algorithms.

Malware Detector	API feature format	Algorithm
MamaDroid [11]	Markov Chain of API Calls	Random Forest
DroidEvolver [12]	API Occurrence	Model Pool
Drebin [3]	Selected API Occurrence	SVM
Drebin-DL [9]	Selected API Occurrence	DNN
Ceschin et al. [24]	API Frequency	Random Forest

we re-implement them following the descriptions in their papers. Note that some of these works may have several configurations. In this situation, we select the best-performing one. For example, for MamaDroid we use its "package mode" and the random forest algorithm, following previous works [16], [19]. We also test MamaDroid in "family mode" and the results are listed in Appendix B. For other configurations, we strictly follow the original papers and make sure our reproductions can achieve the results as stated in their papers.

Enhancement with APIGraph. We enhance these baselines by transforming their usage of APIs in the feature space to leverage the API knowledge and do not change other parts of the original detectors. For example, DroidEvolver uses the occurrence of APIs to represent the feature vector of a sample while the enhanced DroidEvolver (w/ APIGraph) uses the occurrence of API clusters as its feature vector. Following this way, a malware detector can be enhanced with most of its parts untouched and therefore can be directly deployed to replace the original one. For example, Drebin claims that it is lightweight and thus can work on mobile devices, our enhanced version also has this capability.

6 Evaluation

In this section, we evaluate how APIGraph helps to slow down the aging of state-of-the-art malware detectors. Specifically, the experiments are designed from the following aspects: ❶ *prolonging model lifetime* (§6.1), ❷ *reducing maintaining efforts* (§6.2), ❸ *stabilizing feature space* (§6.3), ❹ *capturing API closeness* (§6.4), and ❺ *robustness against adversarial attacks* (§6.5).

6.1 Prolonging Model Lifetime

Metrics: To evaluate how APIGraph can help prolong the lifetime of existing models, we use the *AUT* metric proposed by Tesseract [16], which is the *A*rea *U*nder the curve during a certain *T*ime, as shown in Equation 1.

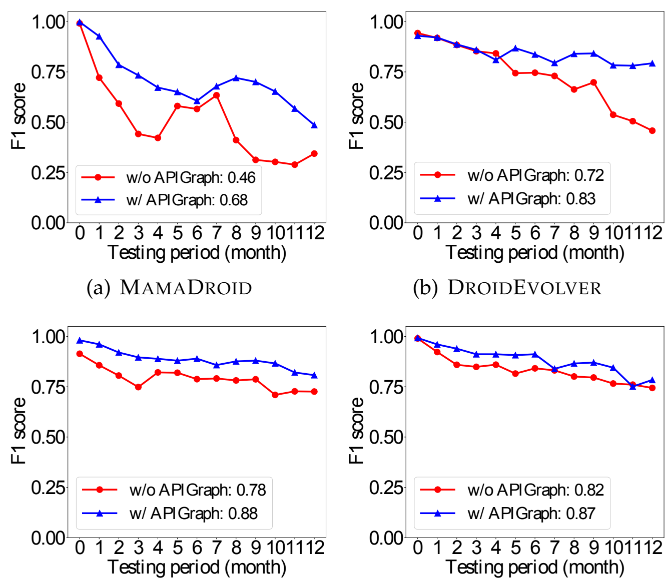
$$AUT(f, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{[f(k+1) + f(k)]}{2} \quad (1)$$

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 the performance metric evaluated at the time k. In our experiments, the final metrics for Android and Windows platforms are AUT(F_1, 12m) and AUT(F_1, 6y) respectively, which are the F_1 score across 12 months and F_1 score across 6 years. An AUT metric that is closer to 1 means better performance over time.

TABLE 7
AUT(F_1, 12m) of evaluated Android malware detectors before and after leveraging API relation graph. For each testing year, the detectors 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 detector without APIGraph, i.e. the original detector.
[2] w/ denotes the detector enhanced with APIGraph.



(a) MamaDroid (b) DroidEvolver (c) Drebin (d) Drebin-DL

Fig. 6. AUT(F_1, 12m) of evaluated Android malware detectors before and after leveraging API relation graph. Each detector is trained in 2012 and tested in 12 months of 2013. Note that month 0 indicates the time when the detector is initially trained.

Experimental Settings: Considering the dataset scales, we test Android detectors monthly and Windows yearly. For Android detectors, we train a model on a particular year (say 2012), and sequentially test its performance on 12 months of the next year (i.e. 2013), and then calculate AUT(F_1, 12m), before sliding to the next train-testing year pair (i.e. training on 2013 and test on 2014). Note we only test the performance of a model over a year because many models age significantly so that they become unusable after one year. For Windows detectors, we train a model on samples of 2012, and test the model on each year from 2013 to 2018, and calculate AUT(F_1, 6y). Because Windows samples in each year do not naturally satisfy the spatial constraint, we randomly down-sample malware to 10% and test 50 times and average the results. For each malware detector on each platform, we evaluate its performance with and without enhancing by APIGraph.

Results: Table 7 shows the AUT(F_1, 12m) values of four Android detectors 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

TABLE 8
F_1 and AUT of evaluated Windows malware detector before and after leveraging APIGRAPH.

	2013	2014	2015	2016	2017	2018	AUT
w/o APIGraph	0.868	0.915	0.794	0.679	0.906	0.791	0.825
w/ APIGraph	0.895	0.941	0.867	0.804	0.911	0.800	0.874

for the four detectors, which indicates that APIGRAPH is capable of slowing down model aging, i.e. prolonging the lifetime of malware detectors. We also breakdown the results into months and show the F_1 score of four malware detectors which are tested in 2013 and trained with samples of 2012 in Figure 6. As shown in this figure, APIGRAPH successfully slows down the performance degrading of all enhanced malware detectors.

Tabel 8 shows the F_1 score of every test year and AUT(F_1, 6y) of the Windows detector. Compared to Android detectors, the Windows detector ages relatively slowly, indicating that Android malware may evolve more actively than Windows ones. Nevertheless, APIGRAPH still helps improve the performance of the original detector. Specifically, the AUT has been increased from 0.825 to 0.874, a 5.9% improvement. We also note that in the year 2016, the F_1 of the original detector drops drastically, while with the help of APIGRAPH it can still achieve a relatively good result.

Findings: APIGRAPH enhances the sustainability of tested models by 5.9% to 19.6%, indicating that it can significantly prolong the lifetime of existing malware detectors under evolved malware samples.

6.2 Reducing Maintaining Efforts

Metrics: The purpose of this experiment is to find out how many human efforts APIGRAPH can save while maintaining a high-performance malware detector. Specifically, the comparison adopts two metrics: (i) the retraining frequency, and (ii) the number of malware to label.

Experimental Settings: First, we train a model and test it month by month (or year by year for the Windows detector). Then, when the F_1 score of a detector falls below a low threshold T_l, we retrain the model so that it can reach a higher threshold T_h. We calculate how many human efforts (i.e. the above two metrics) are needed in the retraining step. To retrain an aged model, we adopt the active learning [16] method, which is an optimization to normal retraining methods. Specifically, we use the *uncertain sampling* [16] algorithm to actively select the most uncertain predictions. In detail, first we select the most 1% uncertain samples to retrain the detector, and then gradually increase the percentage by 1% until the F_1 score reaches T_h. Through this way, we can figure out the minimum efforts to maintain a high-performance model.

In this experiment, the detectors are initially trained on all the apps in 2012. We then adopt the above approach to maintain the performance of the detectors from Jan 2013 to Dec 2018, and observe the retrain frequency and the number of malware to label.

Results: Table 9 shows the retrain frequency and the number of malware to label from 2013 to 2018 with (T_l = 0.8, T_h = 0.9) for Android and (T_l = 0.85, T_h = 0.9) for Windows.

TABLE 9
Retrain efforts for tested models, in terms of retraining frequency (months between two retrains) and the number of labeled samples, when using active learning with thresholds (T_l = 0.8, T_h = 0.9) for Android and (T_l = 0.85, T_h = 0.9) for Windows detectors.

	retrain frequency			# labeled samples		
	w/o [1]	w/ [2]	Improves	w/o	w/	Improves
MAMADROID	1.6	2.1	31.25%	22,411	14,999	33.07%
DROIDEVOLVER	3.8	4.8	26.32%	20,767	12,913	37.82%
DREBIN	1.3	5.5	323.08%	167,005	6,173	96.30%
DREBIN-DL	2.8	4.5	60.71%	28,408	9,292	67.29%
Windows [3]	3	6	100%	471	252	46.50%

[1] w/o denotes the detector without APIGraph, i.e. the original detector.
[2] w/ denotes the detector enhanced with APIGraph.
[3] The retrain frequency for Windows is in years.

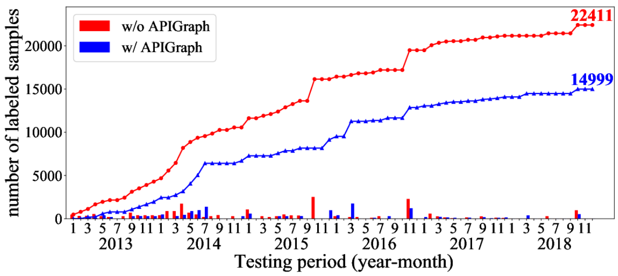
The experiments for Windows are conducted year by year and use 0.85 as the low threshold, as the Windows detector ages relatively slowly. APIGRAPH can improve the retrain frequency by 31.25%, 26.32%, 323.08%, 60.71%, and 100%, while save the number of samples to label by 33.07%, 37.82%, 96.30%, 67.29%, and 46.50% respectively. Especially for DREBIN, it used to retrain the model every 1.3 months, but after enhanced with APIGRAPH, it only needs to retrain the model every 5.5 months, and the labeling efforts drop from 167K to about only 6K. We also count the samples to be labeled in both cumulative and monthly/yearly distribution numbers, to visually show how APIGRAPH can help reduce maintaining cost, as in Figure 7.

Findings: APIGRAPH can reduce retrain frequency by 26.32% to 323.08%, and reduce the numbers of manually-labeled samples by 33.07% to 96.30%, indicating that it can significantly reduce human efforts when maintaining various malware detectors.

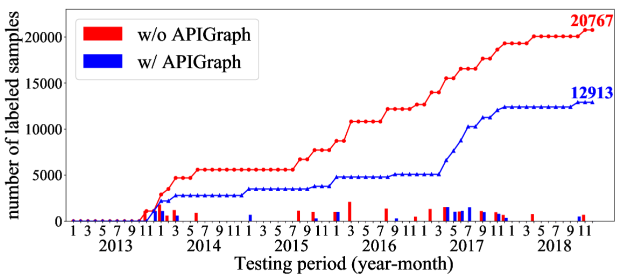
6.3 Stabilizing Feature Space

Metrics: We observe that one drawback of using individual APIs is that malware evolution can disturb the stability of the feature space using different API implementations. In this experiment, we want to evaluate how API clustering helps stabilize the feature space of different malware variations. To do this, we first sort all the malware samples in one family by their appearing time and then divide them into 10 groups so that each group contains 10% samples of this family. The appearing time of all samples in one group is strictly ahead of samples from the next group. Then we calculate a feature stability score of every two adjacent groups using the Jaccard similarity coefficient: $J(A,B) = |A \cap B|/|A \cup B|$, 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 malware groups.

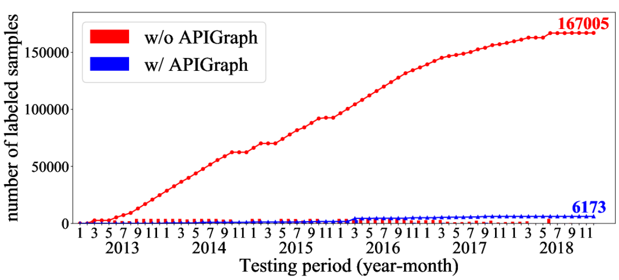
Experimental Settings: We first download 109,770 malware samples from different malware repositories as described in §5.1, and use a malware labeling tool named Euphony [51] to get their family labels. We only keep those malware that Euphony can get reliable labels, which leaves 101,360 malware from 1,120 families. Then we select the 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).



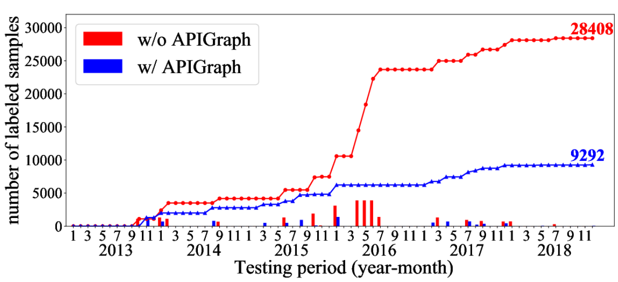
(a) The efforts in sample labeling for MAMADROID



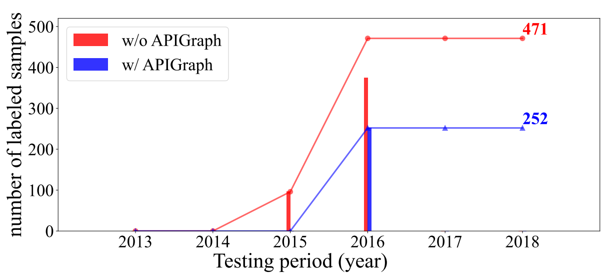
(b) The efforts in sample labeling for DROIDEVOLVER



(c) The efforts in sample labeling for DREBIN



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



(e) The efforts in sample labeling for the Windows detector

Fig. 7. The number of malware samples to label using active learning with fixed retrain thresholds (T_l = 0.8, T_h = 0.9) for Android and (T_l = 0.85, T_h = 0.9) for Windows. Each bar shows the number of samples labeled in that month, while each curve shows the cumulative number.

The top 30 families, as well as the number of their samples, are listed in Table 10 of Appendix A.

We then use static analysis with the help of *apktool* [52] to disassemble malware code and obtain API features. For each malware family, we calculate the feature stability score from two perspectives: using individual APIs as the features and using API clusters as the features.

Results: Figure 8 shows the distribution of feature stability scores for each malware family with API and API clusters as features. We can see 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 features directly. This explains why APIGRAPH can help models capture malware evolution, as malware developers tend to use semantically similar APIs to implement the same or similar functionalities.

Findings: APIGRAPH successfully captures semantic similarity among evolved malware samples in a family.

6.4 Capturing API Closeness

Metrics: In this experiment the t-SNE [53] method is used to project and visualize all the APIs into a two-dimensional space.

Experimental Settings: We get the API embeddings from the API relation graphs for both Android and Windows platforms, and feed these embeddings into the t-SNE algorithm from sklearn [40].

Results: Figure 9 demonstrates parts of the visualization graph for both Android and Windows APIs. More specifically, figure 9(a) shows the results on Android APIs, where APIs from the 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 abstraction method, like that adopted by MAMADROID, is inaccurate in capturing semantics information. Similarly, the Windows APIs are well separated into different clusters, as shown in figure 9(b). For example, file-related APIs which are from different header files, such as *winbash.h->MoveFile*, *fileapi.h->LockFile*, are close to each other.

Findings: Semantically close APIs are grouped in the same or close cluster in the embedding space by APIGRAPH.

6.5 Robustness against Adversarial Attacks

Metrics: This paper focuses on currently the most common and practical way of malware evolution, i.e. using alternative APIs to implement similar malicious functionalities. Considering that adversarial attacks are becoming one important way of evolution and evasion, we also test how APIGRAPH can improve the robustness of existing malware detectors against adversarial attacks. We use evasion rate and number of changed features as the robustness metrics, as in [54]. Specifically, the evasion rate ER is calculated by $ER = FN/P$.

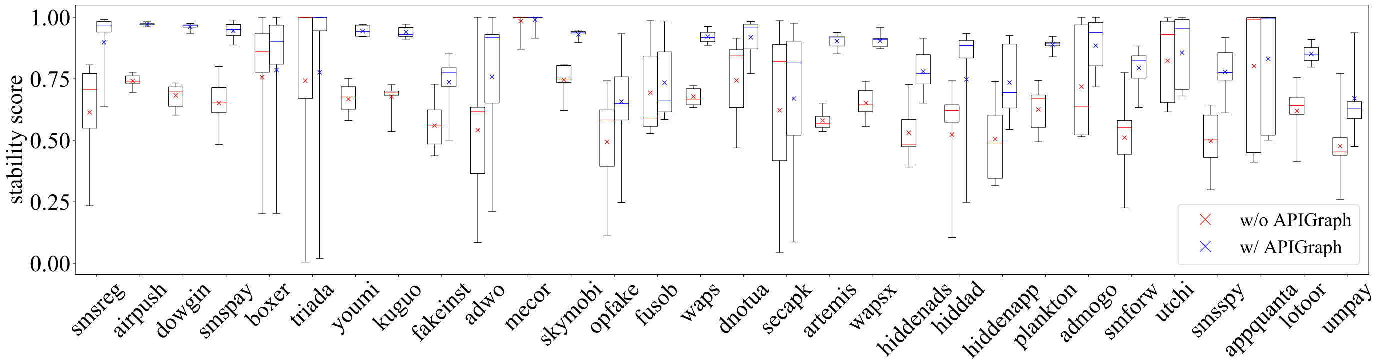
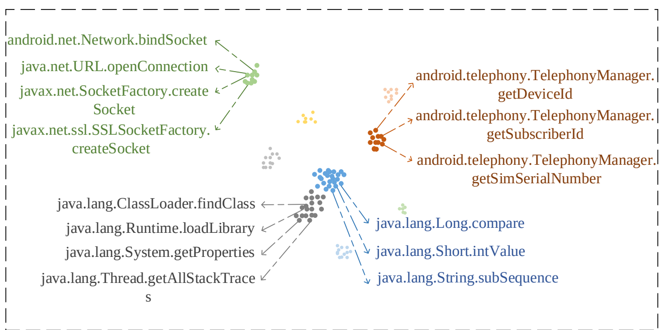
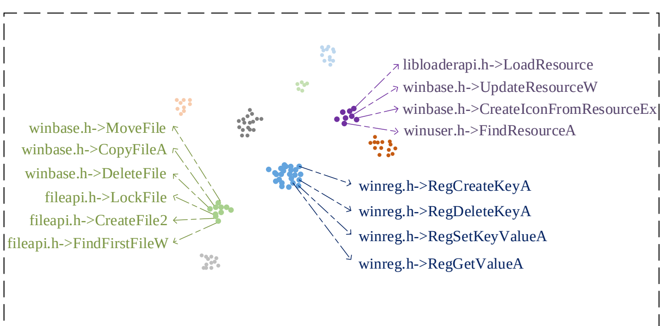


Fig. 8. The distribution of feature stability scores for every top 30 malware family, when considering APIs as features and API clusters as features.



(a) Visualizing Android APIs in the motivating example and the "java.lang" package



(b) Visualizing Windows APIs about file, regedit, and resource

Fig. 9. Projecting API embeddings into a two-dimensional space through t-SNE and visualizing them.

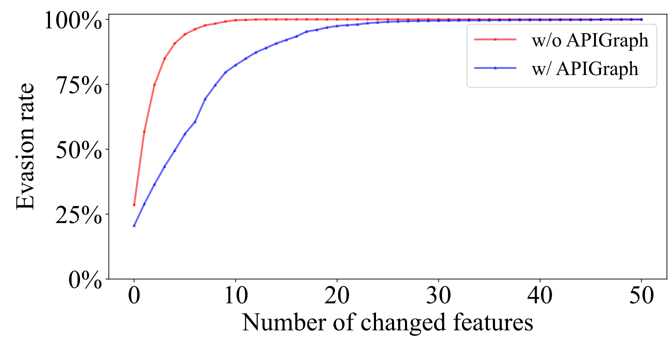


Fig. 10. White-box adversarial attacks against original and enhanced DREBIN-DL.

Experimental Settings: We use the white-box adversarial attack proposed by the DREBIN-DL paper [9]. DREBIN-DL uses multilayer perceptron (MLP) as the classification algorithm, and in [9] they choose the perturbation with the maximal positive gradient to generate feature-space adversarial examples. The attack is conducted on both the original and enhanced DREBIN-DL, with 32,089 malware in Table 5. Specifically, a model is trained on each year (say 2012) and attacked using malware from the next year (i.e. 2013), until all malware are tested. During the adversarial attacks, we record how many features are needed to change for one malware to successfully evade the detectors.

Results: We draw the CDF of all number of changed features in Figure 10, where the y-axis represents the evasion rate. We can see that DREBIN-DL with the help of APIGRAPH has a lower evasion rate than the original one under the same circumstances. Also, it needs to change 12 features to evade original DREBIN-DL for 99.9% malware samples, while 30 features for the enhanced model. Note that more features changed means that the attacker needs more cost and the defender has a better chance to detect the attack.

Findings: APIGRAPH can help improve the robustness of existing malware detectors against adversarial attacks.

7 DISCUSSION AND LIMITATION

Data-perspective Methods VS. Feature space-perspective Methods. To tackle the model aging of machine learning models in malware detection, methods from two different perspectives are proposed. Data-perspective methods, including retraining [21], online learning [10], [12] and active learning [16] try to learn more statistical properties from new data samples, thus to better detect emerging malware; While feature space-perspective methods focus on leveraging the domain-specific knowledge to guide the machine learning models to capture the intrinsic properties of the underlying tasks. Most of the previous works focus on data-perspective methods, while this paper makes the first step to incorporate API knowledge to help models from the feature space perspective. By applying the idea to different platforms and models, this paper proves the effectiveness of feature space-perspective methods themselves, as well as combining with data-perspective methods. We believe the combination of the two methodologies is necessary to build better malware detectors, and also other security applications such as malware classification, anomaly detection, etc.

API Documentation VS. Other Knowledge Sources. In this paper, we focus on the API reference documentation to extract API knowledge 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. Nevertheless, other sources, such as the source code, developing tutorial, and developer guides may also contain intrinsic knowledge that are unlikely obtained from data samples but are useful to machine learning models. In our future study, we will consider knowledge from these sources and use them to help existing models.

Non-API-based Malware Detectors. APIs are a popular type of features widely adopted by many other existing malware detectors [1], [6], [55], [56], mainly because APIs are essential in implementing malware functionalities. There indeed are two types of detectors that do not directly adopt APIs as a feature. First, some detectors, e.g., Mclaughlin et al. [8], 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 detectors by transforming those opcodes to incorporate API cluster information. Second, some detectors, e.g., MassVet [57], mainly adopt UI structures for malware detection. Such detectors may age quickly given malware evolution because those features like UI structures are unreliable and easy to change.

Malware Obfuscation. Obfuscation techniques, such as reflection, packing [58], and dynamic code loading [59] may be used to bypass existing analysis, especially feature extraction. We believe this is an orthogonal problem to what has been studied in APIGRAPH. In future works, solutions [60], [61], [62], [63] focusing on malware obfuscation can be used to help extract features from software.

8 RELATED WORK

8.1 Malware Detection

Malware detection has been an active research area over the past years. Recently, learning-based methods are becoming increasingly popular. In these works [1], [2], [3], [4], [6], [11], [55], [56], [64], [65], [66], APIs are commonly used as the features to detect malice. Specifically, DroidAPIMiner [1] and DREBIN [3] use the occurrence of APIs; DroidEye [55] and StormDroid [6] use API frequency; MalDolzer [56], Ki et al. [65], and Amer et al. [66] adopt API calling sequences; and DroidMiner [2], DroidSift [4], and AppContext [64] adopt API call graph.

Most of the above works treat each API separately and ignore the inherent relations among these APIs. MAMADROID [11] is one of the few exceptions that group APIs according to their packages or families. However, packages and families only reflect the hierarchy relations between APIs. In contrast, the semantic groups used in APIGRAPH can better capture the semantic relations between APIs, which can be more accurate in describing malware evolution, as shown in § 6.4.

8.2 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 [12], or time decay [16], [17], or model degradation [18] and deterioration [19] in the literature. Transcend [21] proposes to use statistic techniques to detect concept drift before the model's performance starts to fall sharply. Tesseract [16] proposes a new metric named AUT to effectively measure how a model performs over time in the setting of concept drift. EveDroid [18] and DroidSpan [19] 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 detectors.

Previous works in the field of malware analysis also notice model aging. Gianni et al. [25] point out that malware may remove, replace, or add useless API calls to evade analysis. Thus they propose an association rule-based approach that extracts nonadjacent and representative subsequences to tolerate useless API modification. Apart from useless API calls, APIGRAPH can also tolerate critical API call modification, as long as these APIs share close semantics. Ficco et al. [23] propose combining diverse and stochastic detectors, which can effectively improve the resiliency against determined adversaries. This strategy can be used to work together with APIGRAPH, where API knowledge are used to enhance each combined detector. APIGRAPH is orthogonal to existing learning-based approaches, such as retraining, active learning and ensemble learning, etc. When combined with these methods, APIGRAPH can help develop better detectors that are more resilient to concept drift.

8.3 Knowledge Graph and API Knowledge

Knowledge graphs [67], [68] have been successfully constructed and applied to many real-world tasks, such as extracting information and answering questions. Inspired by the concept of the knowledge graph, we propose an API relation graph to represent the internal relations among diverse programming entities. The major challenges here are that we need to extract and represent platform-specific entities and relations. Several knowledge graph embedding algorithms have been proposed, including TransE [34], TransH [35], and TransR [36]. Our API embedding algorithm uses the TransE with some variations to convert APIs in the relation graph to embeddings.

API reference documents contain abundant information about APIs. Maalej et al. [27] have developed a taxonomy of knowledge types in API reference documents. Based on this taxonomy, Li et al. [69] 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.

9 CONCLUSION

Malicious software keep evolving over time to avoid being detected, leading to model aging of ML-based malware detectors. Most existing works try to mitigate model aging from the data perspective, i.e. labeling new samples and retraining the aged models. However, data-perspective methods often need huge efforts and the updated models are still blind of the root cause of malware evolution. In this paper, we observe that one common way of malware evolution is to change the implementation while preserving the same maliciousness logic, for example, using interchangeable APIs. Therefore, we propose to let ML models capture the semantic similarity among APIs, called *API knowledge*, to better detect evolved malware. We propose a general framework, named APIGRAPH, that can help extract API knowledge from documents and leverage these knowledge to enhance existing malware detectors for both Android and Windows platforms.

We applied APIGRAPH on 5 SOTA malware detectors and evaluate them on large-scale, evolutionary datasets. Extensive experiments show that APIGRAPH can significantly slow down the aging and reduce maintaining efforts for these detectors. We have publicly released our datasets and source code at https://github.com/seclab-fudan/APIGraph to facilitate researches in this area.

ACKNOWLEDGMENTS

We would like to thank all reviewers for their helpful comments. This work was supported in part by the National Natural Science Foundation of China (62102091, U1636204, U1836210, U1836213, U1736208, 61972099), China Postdoctoral Science Foundation (BX2021079, 2021M690706), 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.

REFERENCES

[1] Y. Aafer, W. Du, and H. Yin, "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, 2013, pp. 86–103.

[2] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "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, 2014, pp. 163–182.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014, pp. 23–26.

[4] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "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, 2014, pp. 1105–1116.

[5] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "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, 2015, pp. 422–433.

[6] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "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, 2016, pp. 377–388.

[7] S. Hou, A. Saas, L. Chen, and Y. Ye, "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, 2016, pp. 104–111.

[8] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé *et al.*, "Deep android malware detection," in *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 2017, pp. 301–308.

[9] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 2017, pp. 62–79.

[10] A. Narayanan, L. Yang, L. Chen, and L. Jinliang, "Adaptive and scalable android malware detection through online learning," in *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2016, pp. 2484–2491.

[11] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[12] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving android malware detection system," in *2019 IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE, 2019, pp. 47–62.

[13] E. Raff, J. Sylvester, and C. Nicholas, "Learning the pe header, malware detection with minimal domain knowledge," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. ACM, 2017, pp. 121–132.

[14] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware detection by eating a whole exe," 2018. [Online]. Available: https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422/15577

[15] H. S. Anderson and P. Roth, "Ember: an open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.

[16] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating experimental bias in malware classification across space and time," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 729–746.

[17] Y. Miao, C. Chen, L. Pan, Q.-L. Han, J. Zhang, and Y. Xiang, "Machine learning based cyber attacks targeting on controlled information: A survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–36, 2021.

[18] T. Lei, Z. Qin, Z. Wang, Q. Li, and D. Ye, "Evedroid: Event-aware android malware detection against model degrading for iot devices," *IEEE Internet of Things Journal (IOTJ)*, 2019.

[19] H. Cai, "Assessing and improving malware detection sustainability through app evolution studies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 2, 2020.

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

[21] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *Proceedings of 26th USENIX Security Symposium (USENIX Security)*, 2017, pp. 625–642.

[22] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of android malware detection with deep neural models," *ACM Computing Surveys (CSUR)*, vol. 53, no. 6, pp. 1–36, 2020.

[23] M. Ficco, "Malware analysis by combining multiple detectors and observation windows," *IEEE Transactions on Computers*, 2021.

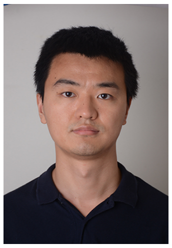
[24] F. Ceschin, F. Pinage, M. Castilho, D. Menotti, L. S. Oliveira, and A. Gregio, "The need for speed: An analysis of brazilian malware classifers," *IEEE Security & Privacy*, vol. 16, no. 6, pp. 31–41, 2018.

[25] G. D'Angelo, M. Ficco, and F. Palmieri, "Association rule-based malware classification using common subsequences of api calls," *Applied Soft Computing*, vol. 105, p. 107234, 2021.

[26] Trend Micro, "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/, 2018.

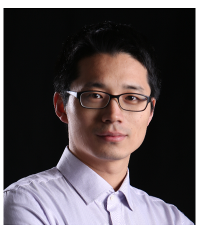
[27] W. Maalej and M. P. Robillard, "Patterns of knowledge in api ref-

erence documentation," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 9, pp. 1264–1282, 2013.
[28] Android Developers, "Android API Reference," https://developer.android.com/reference/, 2020.
[29] Android Developer, "Android Version Distribution Dashboards," https://developer.android.com/about/dashboards, 2021.
[30] Windows Developers, "Windows API Documents," https://docs.microsoft.com/en-us/windows/win32/api/, 2020.
[31] Android Developers, "Manifest.permission," https://developer.android.com/reference/android/Manifest.permission, 2020.
[32] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "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*, 2020, pp. 757–770.
[33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 3111–3119.
[34] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Proceedings of the 26th Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 2787–2795.
[35] Z. Wang, J. Zhang, J. Feng, and Z. Chen, "Knowledge graph embedding by translating on hyperplanes," in *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI)*, 2014.
[36] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu, "Learning entity and relation embeddings for knowledge graph completion," in *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI)*, 2015.
[37] M. Syakur, B. Khotimah, E. Rochman, and B. Satoto, "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, no. 1. IOP Publishing, 2018, p. 012017.
[38] spaCy, "spaCy - Industrial-Strength Natural Language Processing," https://spacy.io/, 2020.
[39] TensorFlow, "TensorFlow - An End-to-end Open Source Machine Learning Platform," https://www.tensorflow.org/, 2020.
[40] scikit-learn, "scikit-learn, Machine Learning in Python," https://scikit-learn.org, 2020.
[41] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 217–228.
[42] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Revisiting android permission specification analysis," in *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, 2016, pp. 1101–1118.
[43] VirusTotal, "VirusTotal," https://virustotal.com, 2020.
[44] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, and G. Wang, "Measuring and modeling the label dynamics of online anti-malware engines," in *Proceedings of the29th USENIX Security Symposium (USENIX Security)*, 2020.
[45] Université du Luxembourg, "AndroZoo," https://androzoo.uni.lu/, 2016.
[46] VirusShare, "VirusShare," https://virusshare.com, 2020.
[47] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Bonn, Germany: Springer, 2017, pp. 252–276.
[48] H. X. Li Y, Jang J, "AMD Dataset," http://amd.arguslab.org/sharing, 2019.
[49] MamaDroid, "MamaDroid code," https://bitbucket.org/gianluca_students/mamadroid_code/src/master/, 2021.
[50] DroidEvolver, "DroidEvolver code," https://github.com/DroidEvolver/DroidEvolver, 2021.
[51] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro, "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, 2017, pp. 425–435.
[52] Apktool, "A Tool for Reverse Engineering Android APK Files," https://ibotpeaches.github.io/Apktool/, 2019.
[53] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.
[54] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ml attacks in the problem space," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1332–1349.
[55] L. Chen, S. Hou, Y. Ye, and S. Xu, "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, 2018, pp. 782–789.
[56] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.
[57] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "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)*, 2015, pp. 659–674.
[58] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "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)*, 2018.
[59] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi, "Grab'n run: Secure and practical dynamic code loading for android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015, pp. 201–210.
[60] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 318–329.
[61] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. D. Ernst *et al.*, "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, 2015, pp. 669–679.
[62] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed android applications," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 2015, pp. 293–311.
[63] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "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, 2014, pp. 23–26.
[64] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE Press, 2015, pp. 303–313.
[65] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on api call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 659101, 2015.
[66] E. Amer and I. Zelinka, "A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence," *Computers & Security*, vol. 92, p. 101760, 2020.
[67] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: A collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008, pp. 1247–1250.
[68] Google, "Google - Introducing the Knowledge Graph: Things, Not Strings," https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html, 2020.
[69] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 183–193.

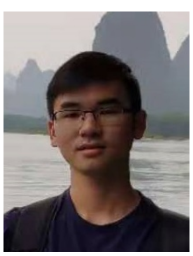


Xiaohan Zhang is currently a postdoctoral researcher at Fudan University. He obtained B.Eng and Ph.D. from Fudan University, under the supervision of Prof. Min Yang. His research interests include malware detection, mobile security, and AI security.

Mi Zhang received the Ph.D. degree in computer science from University College Dublin in 2010. She is currently an associate professor in the School of Computer Science, Fudan University. Her research interests include theoretical and applied machine learning.

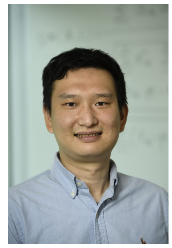


Yuan Zhang received the B.Eng. degree from Nanjing University in 2009 and the Ph.D. degree from Fudan University in 2014, where he is currently an associate professor with the Software School. His research interests include system security and compiler techniques.



Ming Zhong received the B.Eng. degree from Jinan University in 2019. He is currently pursuing a master's degree from Fudan University, Shanghai, China. His research interests include mobile security and machine learning.

Xin Zhang is currently an undergraduate student majoring in Information Security in the School of Computer Science, Fudan University. Her research interests include system security and malware detection.



Yinzhi Cao is an assistant professor in Computer Science at Johns Hopkins University. He earned his Ph.D. in Computer Science at Northwestern University and his B.E. degree in Electronics Engineering at Tsinghua University in China. His research mainly focuses on the security and privacy of the Web, smartphones, and machine learning. His past work was widely featured by media outlets, such as NSF Science Now, CCTV News, and IEEE Spectrum. He received two best paper awards at SOSP'17 and IEEE CNS'15.

Min Yang received the B.Sc. and the Ph.D. degrees in computer science from Fudan University in 2001 and 2006, respectively, where he is currently a professor in the School of Computer Science. His research interests include system security and AI security.