API2Vec: Learning Representations of API Sequences for Malware Detection

Lei Cui*
Zhongguancun Laboratory
Beijing, China
cuilei@iie.ac.cn

Jiancong Cui*
School of Cyber Security University of Chinese Academy of Sciences
Institute of Information Engineering at Chinese Academy of Sciences
Beijing, China
cuijiancong@iie.ac.cn

Yuede Ji
University of North Texas
Texas, USA
yuede.ji@unt.edu

Zhiyu Hao†
Zhongguancun Laboratory
Beijing, China
haozy@zgclab.edu.cn

Lun Li
Institute of Information Engineering at Chinese Academy of Sciences
Beijing, China
lilun@iie.ac.cn

Zhenquan Ding
Institute of Information Engineering at Chinese Academy of Sciences
Beijing, China
dingzhenquan@iie.ac.cn

ABSTRACT

Analyzing malware based on API call sequence is an effective approach as the sequence reflects the dynamic execution behavior of malware. Recent advancements in deep learning have led to the application of these techniques for mining useful information from API call sequences. However, these methods mainly operate on raw sequences and may not effectively capture important information especially for multi-process malware, mainly due to the *API call interleaving problem*.

Motivated by that, this paper presents API2Vec, a graph based API embedding method for malware detection. First, we build a graph model to represent the raw sequence. In particular, we design the temporal process graph (TPG) to model inter-process behavior and temporal API graph (TAG) to model intra-process behavior. With such graphs, we design a heuristic random walk algorithm to generate a number of paths that can capture the fine-grained malware behavior. By pre-training the paths using the Doc2Vec model, we are able to generate the embeddings of paths and APIs, which can further be used for malware detection. The experiments on a real malware dataset demonstrate that API2Vec outperforms the state-of-the-art embedding methods and detection methods for both accuracy and robustness, especially for multi-process malware.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; **Software security engineering**.

*Both authors contributed equally to this research.
†Corresponding author.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598054

KEYWORDS

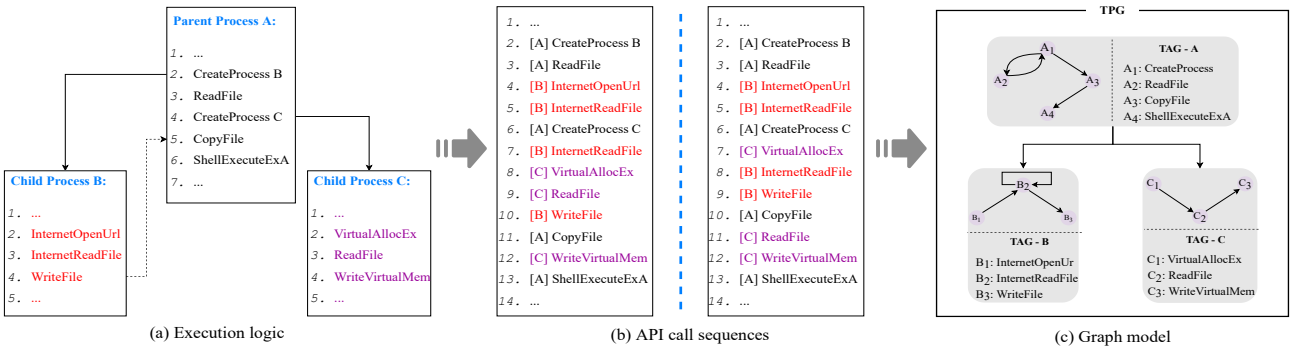
Malware Detection, Embedding, Deep Learning, Random Walk

ACM Reference Format:
Lei Cui, Jiancong Cui, Yuede Ji, Zhiyu Hao, Lun Li, and Zhenquan Ding. 2023. API2Vec: Learning Representations of API Sequences for Malware Detection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598054

1 INTRODUCTION

Malware refer to software that exhibit malicious activities, such as, stealing private information, accessing unauthorized files, and launching attacks [10, 14, 52, 54, 56, 62, 72]. The threats caused by malware have been increasingly dramatically over the years. According to McAfee, there were up to 688 malware threats per minute observed in the first quarter of 2021 [64].

Because of that, malware detection methods are widely deployed to protect the computing devices [47, 54]. Existing methods are mainly classified into two types, i.e., signature-based and behavior-based methods [12, 23, 65]. Signature-based methods usually build a malware signature database by extracting a specific signature pattern (e.g., hash) from known malware [6, 69]. Later, when detecting an unknown software, it would extract its signature following the same method, and match it against the database. Therefore, this method is able to quickly detect known malware with a relatively low false positive rate. However, it is less effective in detecting previously unknown threats, i.e., malware variants or new malware, whose signature may vary a bit [18], thereby suffering high false negatives [12, 51, 73]. Behavior-based detection methods actually run the malware in an isolated environment [48] and extract useful runtime behaviors, such as communication packets [16], API calls [38, 51, 57], and system calls [39, 42, 55]. As a malware would eventually perform some malicious activities, such as, communicating with the controller, downloading additional malware, and accessing privileged files, such behaviors would be able to distinguish malware from goodware. Following that, existing methods



(a) Execution logic (b) API call sequences (c) Graph model

Figure 1: A multi-process malware and its arbitrary API call sequences. a) shows its execution logic. b) shows two sequences of the same malware yet traced at different epochs. c) depicts our graph model, which is robust against various sequences.

usually apply machine learning (ML) methods or deep learning (DL) methods to perform malware detection [27, 68].

1.1 Motivation

API call sequence is one common way to reflect the run-time behavior of a program and has been widely applied in existing malware detection methods [21, 24, 68, 77]. In particular, Gibert et al. [29] and Fan et al. [26] extract features from API call sequences, and apply ML methods (e.g., random forest and gradient boosting) for malware detection. Tran et al. [67] regard the API call sequences as corpus and build a malware detection model with natural language processing (NLP) techniques (e.g., n-gram and term frequency-inverse document frequency). Further, various DL-based methods (e.g., convolutional neural network and long short-term memory) have been applied to the API call sequences for malware detection [74, 77].

However, it is observed that modern malware often utilizes multi-process mechanisms. In a real-world malware dataset containing 14,657 samples from VirusTotal [66], over 60% of them use multiple processes, with some even launching up to 69 processes. The use of multi-process has several purposes, such as improving efficiency or evading detection [61]. Whatever, malware can distribute its behaviors, thus the corresponding API calls, to different processes [13, 22, 28, 34]. Thus, API calls from different processes can interleave with others due to designated execution logic and CPU scheduling, resulting in an API call sequence with arbitrary orders [25, 35], which we name as API interleaving problem. This makes the existing malware detection methods that directly apply techniques to learn features from raw sequences inaccurate, as the raw sequences can be obfuscated by the multi-process mechanism [28]. Therefore, learning on raw API call sequences directly is a significant limitation.

Figure 1 presents an example of a multi-process malware, where a) shows the execution logic of one multi-process malware. Specifically, the main process A forks two child processes B and C that perform different tasks. B is responsible for downloading an executable remotely, which is launched by A later, while C aims to inject code into another process. Logically, A depends on B since it requires the executable of B. C is independent and thus executes concurrently with A or B. As a result, the API call sequence traced by the sandbox is with arbitrary orders. Also, due to the scheduling of C, two sequences traced at different time may vary, as shown in b). Thus, the execution logic is hidden (obfuscated) due to the interleaved API calls and hard to be revealed with existing methods.

1.2 Our Method

We design API2Vec, a graph based API embedding method for malware detection. The main idea is to first use graph modeling to uncover the obfuscated behaviors before implementing detection techniques. More specifically, given the sequence of API calls, we first group the calls based on the processes who initiated them. Then, we build a temporal process graph (TPG) for the whole sequence, where a node denotes a process and an edge denotes the parent-child or child-child relationship. Further, for the APIs inside one process, a temporal API graph (TAG) is built, where a node denotes an API and an edge denotes the happen-before relationship, as shown in Fig. 1(c). Such a representation is able to capture the behaviors of each process and also crossing multiple processes.

With such graphs, we design a heuristic random walk algorithm to mine a number of representative paths. It will traverse the whole TPG and TAGs following 9 behavior and coverage-oriented rules, with the aim to capture more fine-grained malware behavior. By walking inside a TAG, a path is able to accurately capture the intra-process behavior following the execution logic of a single process. Meanwhile, by walking inside the TPG (i.e., across TAGs), the inter-process behavior can also be revealed.

With a corpus of generated paths, we employ Doc2Vec [44] to vectorize the paths. The main advantage of Doc2Vec over commonly used Word2Vec is that it encodes the path (i.e. a set of API calls) as well as a single API at the same time, so that the semantic relationships between different APIs and paths are learned. Since the sequence here is represented by a set of paths, Doc2Vec is naturally suitable for malware analysis. Once the API embedding and paths are available, the ML or DL based methods can be used to perform malware detection.

On a dataset with 14,657 Windows PE malware and 14,113 goodware, where 9,210 and 4,363 are multi-process, respectively, a simple k-NN model with API2Vec achieves 3.47% and 4.78% improvement for *precision* and *recall* over Node2Vec, respectively. Not limited, it achieves better performance when the number of processes increases. This clearly demonstrates the effectiveness of API2Vec. In

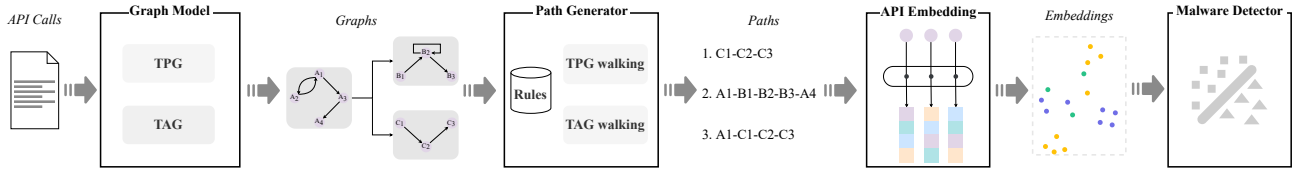


Figure 2: Overview of API2Vec. For the sequence in Fig. 1, one TPG and three TAGs are generated with Graph Model. Then, with Path Generator on graphs, several paths are generated. These paths are then feed into API Embedding module to generate embedding for each path. Finally, the ML models learn on these embeddings for malware detection.

addition, API2Vec is shown to be more robust to both adversarial attacks and concept drift challenges compared with other works.

To summarize, this paper makes three major contributions.

- **Graph-based API sequence representation.** We design temporal process graph and temporal API graph to accurately model the intra-process and inter-process behavior (§4).
- **Behavior and coverage-oriented heuristic random walk.** We devise a heuristic random walk algorithm inspired by program behaviors and path coverage (§5). It is able to extract fine-grained behavior and mine more inherent relationships between the nodes from a graph, with fewer paths.
- **Extensive evaluation.** We have conducted extensive evaluations on a real malware dataset (§7) The results have shown the effectiveness of API2Vec against multi-process malware, and robustness to concept drift and adversarial attacks..

2 THREAT MODEL AND ASSUMPTIONS

In this paper, we mainly focus on the detection of Portable Executable (PE) malware in Windows, which widely affects a large amount of users [7]. The studied malware types include downloader, grayware, worm, backdoor, virus, and rogueware, as will be shown in evaluation (Section 7). We believe our method is generic and can be easily extended to other types, especially for multi-process malware.

We use Cuckoo, a popular sandbox to record the runtime activity of a program, to trace API call sequence of malware [1]. We assume that the sandbox could successfully trace the API calls involved in malicious behaviors, which is a standard threat model for dynamic malware analysis [40, 41, 43]. We assume a malware can run smoothly without being blocked or disturbed by the operating system or other protection software in the sandbox. Meanwhile, each malware runs for two minutes, which is sufficient to enable malicious behaviors observed in recent studies [43].

3 PROPOSED APPROACH

3.1 Approach Overview

As stated before, both intra- and inter-process behaviors are not clearly exposed in the raw sequence due to the API interleaving problem. To address this challenge, we first design a graph model to distinguish the operations within processes meanwhile characterizing interactions between processes. Then, we design an algorithm to capture the intra- and inter-process behaviors. In particular, our proposed API2Vec includes four main components (Fig. 2).

1) **Graph model** aims to accurately characterize behaviors of each process and behaviors crossing multiple processes. Specifically, it represents raw API sequence with a directed multi-graph, consisting of a TPG and multiple TAGs. TPG models inter-process behavior, where the node denotes a process and the edge denotes the parent-child or child-child relationship between nodes. TAG is corresponded to one process and models the intra-process behavior. Its node denotes an API and the edge denotes the happen-before relationship between APIs.

2) **Path generator** generates paths on top of graphs, each of which represents fine-grained program behavior with a set of consecutive API calls. One straightforward method is to traverse the graphs, but this would lead to path explosion problem due to the generally large , leading to massive paths. Therefore, Path Generator employs random walk to address the path explosion problem. The random walk on a TAG mines behavior inside a single process, e.g., Path 1 in Fig. 2,

3) **API embedding** utilizes a neural network model to learn representations of paths and APIs from a large corpus of paths, so that the paths and APIs are expressed by vectors of numerical representations. As the path generator has represented raw API sequence as multiple paths, we then apply the idea of Doc2Vec [44] from natural language processing to learn the embedding of each path. Doc2Vec here not only vectorizes the APIs but also the paths, unlike the widely used Word2Vec model that only vectorizes APIs. Therefore, Doc2Vec fits better to our purpose. By employing Doc2Vec on a large corpus of paths, both the paths and APIs are represented by 64-dimensional embeddings, where semantically similar paths (or APIs) will be placed closer.

4) **Malware detector** takes the embeddings of API call sequences as inputs and builds a ML model for malware detection. Specifically, given a set of API call sequences, they will be converted to a corpus of embedding following the previous three components. During the training process, we further divide the training data into training and validation to search the optimal parameters for ML models, e.g., k of k-nearest neighbors (k-NN). During the inference process, an API call sequence will be processed by the four components and eventually classified as malware or goodware.

3.2 Case Study

We present a case study to compare API2Vec with raw sequences and demonstrate how API2Vec captures the intra- and inter-process behaviors. We believe this is the key to improve the performance of detecting malware, especially for multi-process malware.

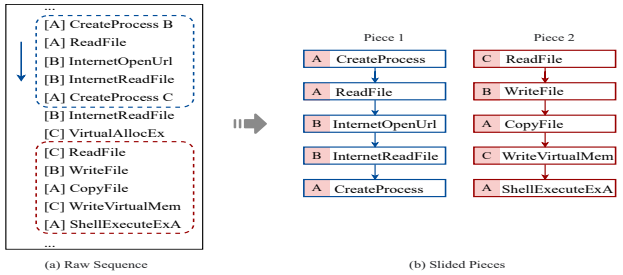


Figure 3: Operation on raw sequences.

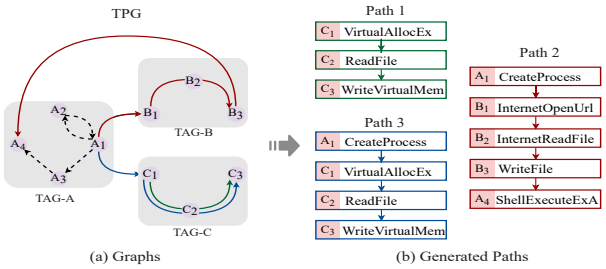


Figure 4: Operation with API2Vec.

3.2.1 Raw Sequence. For the API sequence example in Fig. 1a), if the sequence is separately used, then the intra-process behavior will be learnt easily. However, the inter-process behavior will not be captured since the API call groups are separated. We then examine the performance of two types of methods that operate directly on the raw sequence, as shown in Fig. 3a) (i.e., the left sequence in Fig. 1b)). The first type of methods learn directly on the entire sequence each time, e.g., long short-term memory (LSTM) networks. They have shown good ability mainly due to the capability of learning long-term dependencies. Unfortunately, on the API call sequences that are blended themselves, their learning ability would be compromised. The other type learns on small pieces of sequence within a window, e.g., n-gram, convolutional neural network for text (TextCNN). For this type, we set the window size to 5 and examine two sample pieces as shown in Fig. 3b).

Piece 1. For this piece, the model can probably infer that process *A* created two processes, of which process *B* involves url access and file downloading operations. However, it is difficult for the model to understand the intention of process *A*, which is actually exhibited by ShellExecuteExA lying out of the window.

Piece 2. For this sample, the model would understand that the program created a file and then executed it, but has no way of knowing where the file came from. It is known that downloading payloads remotely is one common operation of malware, which is very important for identifying malware.

From these samples, it can be seen that it is difficult to accurately reveal the program intention from the raw sequences.

3.2.2 API2Vec. As shown in Fig. 4a) (i.e., the graph in Fig. 1c)), it represents the raw sequence with graphs, i.e., one TPG and several TAGs. From the graphs, we examine three sample paths.

Path 1, generated by walking in TAG-C as depicted by the green arrows. It precisely describes the behavior of *C*, i.e., copying data into a newly allocated memory area, which fits process injection.

Path 2, generated by walking over TAG-A and TAG-B as depicted by the red arrows, concatenates API calls from process *A* and *B*. It can more clearly expose the behavior that process *A* creates *B* to download a file remotely and then executes the file, which is valuable for learning its intention. Note that this behavior is difficult to be revealed from the raw sequence.

Path 3, generated by walking over TAG-A and TAG-C as described by the blue arrows, demonstrates that process *C* is launched by *A*. Note that process *B* and *C* are not in parent-child relationship, thus, no edge exists between TAG-B and TAG-C in TPG. Consequently, there is no path where *B* and *C* are interleaved with each other, thereby excluding paths whose behaviors are obfuscated.

From these samples, it can be seen that API2Vec helps reveal the intra-process behavior (TAG) and inter-process behavior (TPG) of program more accurately, which is valuable for malware detection.

3.3 Key Challenges

We identify three key challenges in the implementation of API2Vec.

C1: Modelling temporal and frequency information. The sequence of API calls are not only related logically, but also follows temporal order, i.e., one API occurs after another. In addition, an API can be called repeatedly, which will lead to the number of APIs far fewer than the called times. Therefore, the graph model should be able to represent both temporal and frequency information.

C2: Traversing on hierarchical and multi-graph. The generated graph is hierarchical because the relationship between the processes and APIs is hierarchical, i.e., a process can call multiple APIs. Further, the graph is a multi-graph because there exists multiple edges between two APIs as they might be called at different time. Therefore, one needs to design a graph traversing method that is able to traverse on both hierarchical and multi-graph.

C3: Behavior-aware path generation. Since we will use the paths for API embedding, the generated paths need to capture both the intra- and inter-process behaviors. Therefore, one needs to design a path generation method following program behavior.

We will detail how we address these challenges as follows.

4 TEMPORAL GRAPHS

We first introduce the concept of logical time, a key attribute in our graph model. Then, we detail the definition of TAG and TPG.

4.1 Logical Time

An API call sequence shows the temporal execution order of the APIs called by a program. That means, if an API *a* is ahead of another API *b* in the sequence, then *a* is called earlier than *b*. Note that, this does not mean *a* calls *b* because the caller-callee relationship can not be revealed from the API call sequence. As the temporal API call sequence captures the fine-grained execution behavior, it becomes rather important for malware analysis.

We use logical time, denoted by T, to describe the temporal order of API calls. In particular, T follows the temporal order of API calls in the whole sequence. It starts from 0 and is monotonically increasing

by 1 until the last API call. Thus, each API call is associated with a logical time to denote its order in the sequence.

4.2 Temporal API Graph

Each TAG is associated with one single process. Let S denote the API call sequence of the process, the associated TAG is formalized by $\langle V, E, A\rangle$. V denotes the vertex set, represented by the APIs inside a process, and each vertex $v \in V$ denotes a specific API. E denotes the edge set, where each edge $e \in E$ refers to the temporal order of two vertices. If an API v_d is executed immediately after v_s, then there exists a directed edge e_{sd} from v_s to v_d. We observe there might be multiple edges between two APIs happening at different time. Therefore, we use a multi-graph to represent a TAG, where there can be multiple edges between two vertices.

Of the edges, we use the logical time as the attribute to differentiate them, where the attribute set is denoted as A and logical time set is denoted as T. Specifically, let e^j_{sd} denote the j-th edge between two vertices v_s and v_d, its attribute is expressed by a pair of logical time of two vertices, i.e., $a^j_{sd} = \{t^j_s, t^j_d\}$.

The steps of building a TAG are as follows. First, the logical time is labelled for each API call in the whole sequence, it started from 0 and increases monotonically by 1. Then, the API calls of the same process are grouped. In each group, the first API will be directly added into the TAG as a vertex. Afterwards, the next call is checked whether the associated API is already in the TAG. If not, the API will be added as a new vertex. Then, an edge is attached from the previous API to it and assigned with the associated logical time. This step will execute repeatedly until the sequence traversal is complete, thereby building the TAG for the process.

4.3 Temporal Process Graph

A temporal process graph (TPG) characterizes the whole API call sequence of a program. TPG is formalized by $\langle PV, PE, PA\rangle$. Here, PV denotes the vertices of TPG, and each vertex is a TAG, as shown in Figure 1(c). PE refers to the directed edges between TAGs. Here, there exist two types of edges. First, parent-child edge, which appears when the associated processes of two TAGs are in a parent-child relationship, i.e., one process forks another. It can reveal the process interaction relationship. Second, child-child edge, which appears when API calls of two non-parent-child processes are adjacent to each other. While process interleaving is usually attributed to CPU scheduling, some child processes may interact with each other to carry out malicious behavior. This means that causal relationships may exist between these edges, which would be lost if the edges were removed. Therefore, to uncover these hidden interactions between two processes, we model the child-child edge in our graph model. Similar to TAG, TPG is also a multi-graph where there exist multiple edges between two vertices. If an API of pv_s is adjacent to another API of pv_d, then there is an edge pe_{sd} from pv_s to pv_d. PA denotes the attributes of each edge. Similar to TAG, it is expressed by a pair of the logical time of two TAG vertices, whose associated API calls are adjacent.

We build the TPG out of the whole API sequence following three steps. First, we build TAGs and represent each TAG as a vertex in the TPG. Then, if two TAGs share adjacent API calls, we denote it as an edge labelled with either parent-child or child-child relationship. In this way, the topology of TPG is completed. Finally, for any two API calls that are adjacent in the sequence but across TAGs, an edge is added between the two TAGs accordingly whose attribute is a pair of logical time of the two API calls.

4.4 Discussion

The proposed graph representations brings many benefits. 1) The graph model is robust against various API call sequences from the same program. Therefore, it can be more accurately characterize the program behavior. 2) TAG can exclude the unexpectedly interleaved API calls caused by CPU scheduling since it groups the calls of a single process. 3) TAG expresses the consecutive API calls that have repeatedly appeared in the form of rings. This enables a lot of redundant calls (or behaviors) to be excluded by properly walking these rings. 4) TPG is able to reveal the inter-process behavior as it can help to glue APIs across two TAGs that are in a parent-child relationship. 5) The graph can be traversed directionally rather than blindly (§5) since the edge attributes denote the happen-before relationship.

5 HEURISTIC RANDOM WALK

5.1 Basic Idea of Heuristic Random Walk

5.1.1 Justification of Heuristic Rules. To effectively capture the essential information, we employ random walk to extract paths (i.e., a set of associated APIs) from graphs meanwhile solving the path explosion problem. The naïve random walk, using algorithms such as BFS or DFS, is effective for graphs that showcase spatial structure and are non-hierarchical [20, 31]. However, they perform poorly on our hierarchical and temporal graphs (§7.5). Thus, we need a suitable algorithm that can reveal more fine-grained behaviors with fewer paths by walking on our graphs. To this end, we design 9 heuristic rules to guide the random walk, which fall into three categories. 1) Affinity-oriented rules. Malware often launch actions such as self-propagation and process injection, which are represented in short pieces of associated API calls. This inspires us to generate paths based on API affinity (2 rules). 2) Behavior-oriented rules. Our graph model characterizes intra-process behavior with TAG and inter-process behavior with TPG. Given that a node in graph has many neighbors and there exist multiple edges between two nodes, the walk should follow the program execution to reveal the correct behavior. Hence, we design 5 rules to enable the walk to follow the chronological and logical order of APIs both inside a TAG and across TAGs. 3) Coverage-oriented rules. Some APIs appear frequently in sequences, making the corresponding node a hub in the graph, which further constrains the walk. We design 2 coverage-oriented rules to overcome this issue for discovering more paths and exposing diverse behaviors.

5.1.2 Overflow of Heuristic Random Walk. The proposed graph model is a hierarchical graph, i.e., the node of TPG is TAG, which itself is also a graph with APIs as nodes. Upon such a graph, the heuristic random walk first walks over the TPG, selects a node (i.e., TAG) and then walks on this TAG. After TAG walking is over, it then turns to another TAG and continues walking. More specifically, it traverses the whole graph I times ($I = 10$ by default to reveal more behaviors) and generates $I \cdot N$ paths, where N is the total

number of vertices in TAGs. For each vertex in a TAG, a walker is forked to generate one path for this round of walking. In each round, a current walking epoch t_c is initiated to record the elapsed time of the current walker. It helps to determine the next vertex and indicates whether the walker needs to terminate. Meanwhile, a global path corresponded to the current walker is initiated. Then, the walker starts walking inside the TAG from v_s and generates a path following rules in §5.2, which will be concatenated to the global path. After one TAG walking is completed, the walker will find another TAG with TPG walking to continue TAG walking, as will be detailed in §5.3. Once TPG walking is completed, the generated global path will be put into to a corpus for embedding.

5.2 Random Walk inside TAG

The random walk inside TAG aims to mine the behaviors inside a single process. It starts with the currently visiting vertex v_s at the logical time t_c, repeatedly searches for the next appropriate vertex and appends it to the path, and stops exploring until some conditions are met. Finally, it outputs a path. We tend to answer four research questions upon random walk: 1) which vertex should be selected as the next vertex (vertex selection), 2) which edge of the multiple edges between two vertices will be used (edge selection), 3) when is the walk completed (end conditions), 4) how to return back to the TPG (Return to TPG mechanism).

5.2.1 Vertex Selection. For a currently visiting vertex v_s, all of its neighbor vertices are candidates for walking. Vertices that have a maximum logical time smaller than the current walking epoch t_c expire and are therefore excluded. Of the remaining vertices, the selection as the destination vertex follows three heuristic rules.

***Rule 1**: More edges suggests higher priority* (affinity-oriented). A vertex sharing more edges with v_s implies that these two vertices have a strong affinity in performing actions. Therefore, it is more likely to be selected.

***Rule 2**: Smaller time span denotes higher priority* (behavior-oriented). Time span denotes the minimum difference between the logical time of vertex v_i and the walking epoch t_c. A smaller value implies that v_i will be called soon and more likely to be selected.

***Rule 3**: More visits suggests lower priority* (coverage-oriented). A vertex that is frequently visited may denote a hub in the graph. As a result, higher priority will be given to vertices that have been visited less frequently, in order to improve behavior coverage.

With these rules, for a candidate vertex v_i, let f_i denote the number of edges between v_s and v_i, TS_i denote the time span set for v_i and the time span for the jth edge is calculated by $t_i^j - t_c$, n_i denote the number of visits. Then, the walking probability of v_i, i.e., P_i, is calculated by $f_i/(\sqrt{min(TS_i)} + \sqrt{n_i})$. Note that the minimum time span, denoted as $min(TS_i)$, could be much larger than f_i, e.g., the average value of f_i in our dataset is about 4, while $min(TS_i)$ reach dozens or hundreds. In addition, n_i may also reach dozens as the random walk proceeds. Therefore, we use the square root of $min(TS_i)$ and n_i to enable the effect of f_i upon vertex selection. After the probabilities of all candidates are available, the destination vertex is selected following the probability distribution.

5.2.2 Edge Selection. A TAG is a directed multi-graph. Thus, once the destination vertex is selected, the next step is to select an edge. Given that the attribute of an edge is the logical time of vertices, the edge selection is based on the following rule.

***Rule 4**: Smaller time span denotes higher priority* (behavior-oriented). Fine-grained intra-process behaviors are generally exposed within a short time. Thus, the edge with smaller time span indicates close chronological relationships between APIs and should be selected with high probability.

For the jth edge between v_s and v_d, the probability of being selected is calculated by $1/(TS_d^j)$ where TS_d^j denote its time span.

5.2.3 End Condition. The walker will terminate walking in the current TAG and jump to another TAG for the following cases.

Case 1: No candidate vertex is available. If the logical time of the neighbors have expired (i.e., greater than t_c), or R2P (Return to TPG) is required (§5.2.4), then there will be no available candidate vertex. Therefore, the TAG walking will be terminated.

Case 2: The length of currently walked path reaches a pre-defined threshold. We use L to denote the threshold, which is related to the number of edges in a TAG and is set to 100 by default in our experiment. The limitation on length enables to generate many short and diverse paths, thereby covering more potential behaviors. Meanwhile, a large number of repeated API calls can be excluded.

5.2.4 Return to TPG Mechanism. Return to TPG (R2P) allows the walker to jump to another TAG to continue walking, which helps capture the potential inter-process behavior. It is employed when all the neighbor vertices cannot be viewed as candidates. The probability of a vertex not selected as a candidate follows two rules.

***Rule 5**: Larger time span denotes higher probability* (behavior-oriented). A much larger value implies that the two associated API calls are interrupted by multiple API calls from another process, mainly because of two cases. First, there exists interaction between the two processes. For this case, the two API calls and the calls from another process together represent the whole execution behavior. Thus, the walker should jump to another process to capture the complete behaviors. Second, the two APIs are actually called continuously but disrupted by CPU scheduling. As a result, the API calls between them are irrelevant, so that the associated process is less likely to be touched. To distinguish them, we observe that the CPU scheduling is more arbitrary while the interaction is relatively deterministic. Thus, if there exists only one edge with large time span, it belongs to CPU scheduling. Otherwise, it belongs to process interaction. The probability for this rule is computed by $P_{i1} = min(TS_i) * \alpha/(min(TS_i) + L)$. Here, L is used to avoid excessively use of R2P. α is a hyper parameter, it is set to 1 for the first case to inspire R2P, and 0.3 for the second case to avoid R2P.

***Rule 6**: A much larger number of visits suggests higher probability* (coverage-oriented). A much larger value means the associated API is called repeatedly during program execution. We observe the malware may call sensitive APIs, which are failed frequently but will be called repeatedly, such as RegEnumValueW, StartServiceW, and FindWindowW. Observed that a lot of repeated paths will not improve performance, the walker is likely to explore paths in another TAG when experiencing lots of visits. Let n_i denote the number of visits, we employ min-max scaling to compute the probability of R2P for Rule 6, i.e., $P_{i2} = (n_i - n_{min})/(n_{max} - n_{min})$.

With these two rules in mind, the probability that a candidate vertex is not visited is computed by $P_{i1} + P_{i2} - P_{i1} \cdot P_{i2}$. Then, we randomly filter neighbor vertices following probability distribution. Once all the neighbors are excluded, the walker terminates walking in current TAG and turns to TPG walking.

5.3 Random Walk inside TPG

TPG walking is launched when R2P is required or walking inside a TAG is completed. It aims to find another TAG to continue walking. Similar to TAG walking, it first determines a TPG vertex, and then chooses an edge to pinpoint the start vertex in the selected TAG.

5.3.1 Vertex Selection. For a TPG vertex pv_s, the neighbor vertex whose maximum logical time smaller than t_c will be excluded first. Then, for a candidate pv_i, it is selected following the rules below.

***Rule 7**: Child TAG is preferred* (behavior-oriented). A process generally has more interactions with its child process than its sibling process, which results in more inter-process behaviors. As a result, the child TAG is more likely to be explored by the walker.

***Rule 8**: More edges suggest higher priority* (affinity-oriented). Inter-process behaviors encompass operations across processes and are represented by edges between TAGs in our graph. Thus, the TAG with more edges is favored for uncovering these behaviors.

***Rule 9**: Smaller time span denotes higher priority* (behavior-oriented). Similar to rule 4, a smaller time span implies that the API call of a candidate TAG is more chronologically closer to the currently visited API. Thus, the operations of two TAGs may be highly correlated and involved in inter-process behavior.

Let f_i denote the number of edges between pv_s and pv_i, $min(TS_i)$ denote the minimum time span of edges, the probability that pv_i is selected is $f_i \cdot \beta / min(TS_i)$. Here, β is a parameter to determine the priority of a vertex, i.e., 1 for child vertex and 0.3 for others.

5.3.2 TAG Start Vertex Selection. Once the destination TAG pv_d is determined, the next step is to choose a vertex in pv_d where the walker starts from. Generally, a vertex v_i in pv_d that is neighbor to v_s in pv_s is a candidate. However, there may not exist an edge between v_s and v_i, because the existence of edges between two TAGs does not necessarily guarantee that there exist edges between two vertices distributed over TAGs. For this case, we add a virtual edge from v_s in pv_s to v_i in pv_d, and assign it with attribute $\{t_s, t_i\}$, where t_s denotes the current logical time of v_s and t_i refers to as the minimum logical time of v_i. Then, the probability that v_i serves as the start vertex of the TAG is computed by $f_i / min(TS_i)$.

5.3.3 End Condition. If the logical time of any TPG node is smaller than t_c, the walker should terminate. Otherwise, the walker will find another available TPG node to continue walking.

6 LEARNING TASKS

On a corpus of paths generated in §5, we then perform two independent learning tasks, API Embedding and Malware Detection.

API Embedding. API Embedding utilizes a DL model to learn representations of paths and APIs from a large corpus of paths, so that the paths and APIs are expressed by vectorized embedding. We apply Doc2Vec [44] to learn the embedding of each path and API, which is represented by a 64-dimensional vector.

Specifically, let P denote the set of paths. For the i-th path p_i and one API p_i^j in p_i, we first fetch the context APIs within a window of size C, denoted by $\delta = \{p_i^{j-C}, \cdots, p_i^{j-1}, p_i^{j+1}, \cdots, p_i^{j+C}\}$. Then, p_i^j is represented by the embedding of p_i and APIs in δ, i.e., $E(p_i^j) = W \cdot \frac{1}{2C+1} \left(E(p_i) + \sum_{k \in \delta} E(p_i^k) \right)$, where $E(*)$ denotes the embedding of API or path, and W indicates the weight matrix upon learning. The object is to find embedding E for minimizing the objective function, which is the average negative log-likelihood of each API in all paths, denoted by $-\frac{1}{N_p} \sum_{i=0}^{N_p} \frac{1}{N_{pi}} \sum_{j=0}^{N_{pi}} log \mathcal{P}(p_i^j | \delta, p_i)$, where N_p denotes the size of P, N_{p_i} denotes the length of p_i, and $\mathcal{P}$ indicates the probability of context APIs in a path given the current API p_i^j.

Malware Detection. API2Vec first encodes the API call sequence with the pre-trained path embeddings. Since a sequence yields multiple paths, it is represented by the average value of 64-dimensional embeddings of all paths extracted from the sequence, following other graph embedding methods [8, 20, 60]. In this way, it is not sensitive to the order of vectors of paths. Then, on these encoded sequences, we setup three ML models, i.e., k-NN, SVM, and RF, to train a binary classifier for malware detection. During the inference process, the sequence will be encoded into a 64-dimensional embedding and then classified as malware or goodware.

7 EVALUATION

We implement API2Vec on Ubuntu (20.04.2) with Python 3.7. We use the Cuckoo sandbox to trace programs to acquire API call sequences, and employ NetworkX(2.6.3) library to manage the graph. After a corpus of paths are available, we perform two learning tasks with Google Colab [30]. For API embedding, we use the Gensim package(4.1.2) to deploy Doc2Vec on these paths. The Doc2Vec model is set up with epochs to 10, window size to 5, and embedding size to 64. For malware detection, we setup k-NN, SVM, and RF to train binary classifiers. We evaluate API2Vec with an emphasis on answering the following research questions:

RQ1. What is the ability of API2Vec in malware detection (§7.2)? Here, the ability mainly refers to the precision, recall and F1-score.

RQ2. Is API2Vec robust to the concept drift problem [76] for correctly identifying newly appeared malware (§7.3)?

RQ3. Is API2Vec robust to adversarial malware attacks (§7.4)?

RQ4. Is the proposed components effective in enhancing the performance of API2Vec (§7.5)?

RQ5. Is the runtime overhead of API2Vec low enough to make it efficient and suitable for practical scenarios (§7.6)?

7.1 Experimental Setup

7.1.1 Dataset. We notice that some malware dataset are publicly available, e.g., [11, 15, 37, 49, 75]. However, they suffer several problems and thus are not appropriate for dynamic analysis based malware detection, e.g., static features only [11, 37], goodware are missing [15], malware cannot be executed [49], or size is small [75]. In addition, we tend to evaluate API2Vec in terms of concept drift, which requires diverse malware over a long period. To this end, we prepare a dataset ourselves, which consists of 14,657 malware from VirusSign [3] and 14,113 goodware from NSRL [2]. Then, we use Cuckoo to trace the API call sequence of each sample for 2 minutes. We exclude short sequences (length $\leq$ 10) in which the malicious

behavior are unlikely to lie, following the commonly used method in many studies [63]. Finally, the dataset consists of 28,684 programs (each is with its API call sequence), of which 14,653 are malware (spanning 13 malware types) and 14,031 are goodware. The average length of sequence is 412 for malware and 463 for goodware. Unless specified, we use randomly selected 70% of programs as the training set and the remaining 30% as the testing set.

7.1.2 Compared Methods. API2Vec proposes an embedding method for malware detection. Thus, we compare it with two types of works.

First, embedding methods. 1) Word2Vec [50], a classical model in NLP. Here, it learns the API embedding from a set of sequences. 2) DeepWalk [60], a graph-based embedding that learns latent representations of vertices. Here, we employ the random algorithm of DeepWalk on our graphs. 3) Node2Vec [31], a graph-based embedding that uses novel random walk algorithms with Breadth-First Search (BFS) and Depth-First-Search (DFS). We deploy Node2Vec on our graph and name them Node2VecB and Node2VecD respectively. Once the embeddings of APIs are available, we setup ML models for malware detection, as performed in API2Vec.

Second, malware detection methods. 1) API frequency histogram (Frequency for short) [53], encodes the sequence by vectorizing the API frequency and then uses similarity-based ML algorithms for malware detection. Here, we use k-NN upon reproduction. 2) API sequence Markov chain(Markov) [9], constructs cluster transition matrix for goodware and malware and uses the maximum likelihood accumulated transition value to compute the maliciousness of a sample. 3) BiLSTM-based method (BiLSTM) [77], encodes each API call with run-time parameters and then uses Gated-CNN and BiLSTM to detect malware. 4) A variant of BiLSTM (vBiLSTM), first groups API calls by process and then performs BiLSTM on each subgroup. 5) Bert and fastText model (API-Bert) [74], excludes redundant API calls from the sequence, and then employs Bert and fastText respectively for malware detection. 6) API sequence intrinsic features (API-SIF) [46], models API call sequences with semantic chains and uses BiLSTM for detection. 7) DMalNet [45], concatenates features of API calls and arguments, builds a call graph following the order of API calls, and employs graph neural network for detection. 8) CruParamer [17], encodes API calls and runtime parameters together and then applies DNNs for malware detection.

Metrics. We compare the performance of these methods in terms of i) *precision* that denotes the fraction of true malware (i.e., true positives or TP in short) among the predicted malware, ii) *recall* that refers to as the fraction of true malware that were retrieved, and iii) *f1-score* that is the harmonic mean of *precision* and *recall*, where $f1\text{-}score = 2 \cdot precision \cdot recall/(precision + recall)$.

7.2 Malware Detection Ability (RQ1)

We first evaluate these methods on the whole dataset to provide an overall comparison. Then, we evaluate them on multi-process programs where more interactions exist among processes.

7.2.1 Performance on the Whole Dataset. Table 1 compares the results of different models. For embedding methods, the results of SVM and RF are comparable with K-NN (as will be detailed in §7.5.2), therefore, we mainly report the results of K-NN. As can be seen, our proposed API2Vec achieves the best performance. E.g., it reaches up to 99.23% of *precision* and 99.84% of *recall*, 3.10% and 4.25% higher than Word2Vec, 0.7% and 3.55% higher than BiLSTM, proving the effectiveness of the proposed methods.Note that Word2Vec outperforms DeepWalk and Node2Vec, both of which are graph-based models. The reason is as follows. DeepWalk and Node2Vec are designed for spatial graphs and pay more attention to the structural information of the graph. However, in our graph model, logical time is one important attribute to guide the walking direction. In addition, both TAG and TPG are directed multi-graph so that a node can be visited multiple times. Therefore, DeepWalk and Node2Vec fail to learn from our temporal graphs.

Table 1: Performance comparison on malware detection.

Model	TP	FN	TN	FP	precision	recall	f1-Score
Word2Vec	4,202	194	4,040	169	96.13%	95.59%	95.86%
DeepWalk	4,181	215	4,017	192	95.61%	95.11%	95.36%
Node2VecB	4,179	217	4,024	185	95.76%	95.06%	95.41%
Node2VecD	4,178	218	4,030	179	95.89%	95.04%	95.46%
Frequency	4,188	208	4,028	181	95.86%	95.27%	95.56%
Markov	2,429	1,967	2,960	1,249	66.04%	55.25%	60.17%
BiLSTM	4,233	163	4,146	63	98.53%	96.29%	97.40%
vBiLSTM	4,181	215	3,976	233	94.72%	95.11%	94.91%
API-Bert	4,044	352	3,756	453	89.93%	91.99%	90.95%
API-SIF	4,187	209	3,992	217	95.07%	95.25%	95.16%
DMalNet	4,178	218	3,972	237	94.63%	95.04%	94.84%
CruParamer	4,053	343	4,010	199	95.32%	92.20%	93.73%
API2Vec	**4,389**	**7**	**4,175**	**34**	**99.23%**	**99.84%**	**99.54%**

BiLSTM, as a DL-based method, performs better than the other compared methods. However, it is still worse than our proposed API2Vec, e.g., 97.4% compared to 99.54% in term of *f1-score*. Note that API2Vec trains the malware detector with k-NN, which is much simpler than BiLSTM. This suggests that our embedding method contributes a lot to the performance improvement. vBiLSTM extends BiLSTM by grouping API calls of the same process first, with the aim of better capturing the intra-process behavior. Despite this intention, it performs worse than BiLSTM that learns directly from raw sequences. This is due to the fact that grouping the API calls results in a loss of inter-process behavior information, which hurts the detection performance. Note that CruParamer performs worse than BiLSTM, which is opposite to the results reported in their paper [17]. We suspect that this is due to the limited coverage (only 35.24%) of parameters in our dataset by the rules used in CruParamer, which hurts API embedding based on parameter sensitivity and consequently the detection performance. The Markov method performs poorly on our dataset, i.e., only 60.17% of *f1-score*, similar to the result in another study [70]. We suspect that the reason is as follows. Markov clusters 1,165 APIs for constructing the transition matrix. Yet in our dataset, the number of unique APIs is small, i.e., 49, which compromises clustering and transition construction, thereby hurting the detection performance.

7.2.2 Performance on Multi-process Programs. API2Vec aims to solve the API interleaving problem occurred in multi-process programs. Therefore, we evaluate API2Vec in a testing dataset composed of 4,091 multi-process samples (2,782 malware and 1,309 goodware). As can be seen in Table 2, as the number of processes (#Proc) increases, the compared models exhibit an overall decreasing performance. E.g., Word2Vec reaches 98.44% of *f1-score* when

Table 2: F1-score on multi-process programs.

#Proc	2	3	4	5	≥6
Word2Vec	97.04%	98.44%	94.97%	98.39%	96.92%
DeepWalk	96.57%	98.44%	95.13%	97.56%	96.97%
Node2VecB	96.78%	98.57%	94.82%	96.72%	96.97%
Node2VecD	96.78%	98.27%	94.67%	97.56%	96.18%
Frequency	96.83%	98.48%	95.19%	98.39%	96.06%
Markov	68.08%	79.61%	70.73%	80.37%	88.52%
BiLSTM	97.91%	99.39%	95.89%	99.20%	99.22%
vBiLSTM	96.75%	98.53%	89.02%	97.56%	96.92%
API-Bert	91.82%	97.34%	90.52%	99.20%	96.24%
API-SIF	96.68%	98.66%	90.80%	100.00%	97.60%
DMalNet	96.05%	98.15%	91.83%	99.20%	96.88%
CruParamer	95.01%	98.36%	91.59%	96.72%	98.46%
API2Vec	**99.57%**	**99.87%**	**99.85%**	**100.00%**	**99.22%**

#Proc is 3, yet decreases to 96.92% when #Proc ≥ 6. The reason is as follows. The raw sequence consists of interleaved API calls from different processes, and becomes more chaotic as the number of processes increases. Such a sequence leads to obfuscated program behavior, thereby furthers compromising the learning ability of Word2Vec which learns the representation of APIs from the raw sequence.

As expected, our API2Vec gains much better performance than the other models. For example, it reaches up to 99.22% of *f1-score* when analyzing malware with more than 5 processes, only experiencing a slight drop. We contribute the improvement to the graph model, which could more accurately characterize the inter-process behavior. More specifically, for a malware with more processes, the TPG model characterizes the parent-child and child-child relationship between processes (or TAGs) with edges. Upon TPG walking, it prefers to walk a child process, since the two processes together are more inclined to complete a certain task. Consequently, the inter-process behavior is more easily to be revealed. Although DeepWalk and Node2Vec are built on our graph model, they cannot accurately reveal the behavior across processes. API-Bert and API-SIF utilize multiple DL models for malware detection, e.g., Word2Vec, CNN, and BiLSTM for API-SIF. However, it remains challenging for these models to differentiate between inter-process and intra-process behavior from raw sequences, leading to sub-optimal performance.

7.3 Robustness to Concept Drift (RQ2)

Concept drift appears when new sample appear over time, causing the trained model to mispredict the new one [36, 58, 71]. We evaluate the robustness of models on two types of data.

7.3.1 New Malware that Appear in a New Year. We train a binary-classifier on samples (7,479 malware and 4,835 goodware) spanning from 2009 to 2018 in our dataset, and then measure its performance on testing samples in 2019 (247 malware and 183 goodware) and 2020 (575 malware and 265 goodware), respectively.

As shown in Table 3, the compared models experience performance drop with the time. E.g., the *f1-score* of BiLSTM decreases from 89.08% of 2019 to 83.35% of 2020, i.e., a 5.63% drop. This is mainly because i) malware variants are deployed with new techniques and ii) new malware families appear over time. Despite that, API2Vec still achieves high performance in detecting these new malware, i.e., 98.59% and 99.13%, respectively, demonstrating that it

Table 3: F1-score on samples from varying years.

Model	2019	2020	Model	2019	2020
Word2Vec	86.46%	80.04%	vBiLSTM	89.08%	83.35%
DeepWalk	81.82%	78.66%	API-Bert	84.25%	83.95%
Node2VecD	82.64%	77.62%	API-SIF	87.65%	85.94%
Node2VecB	82.50%	77.58%	DMalNet	89.64%	86.78%
Frequency	80.51%	88.95%	CruParamer	86.02%	91.43%
Markov	77.12%	51.82	**API2Vec**	**98.59%**	**99.13%**
BiLSTM	86.02%	91.43%			

Table 4: F1-score on samples from varying types.

Type	virus	backdoor	worm	grayware	downloader
Word2Vec	88.45%	95.87%	93.39%	91.28%	97.03%
DeppWalk	77.72%	95.17%	93.39%	87.50%	96.50%
Node2VecB	77.90%	95.51%	93.54%	90.65%	96.40%
Node2VecD	77.95%	94.99%	93.63%	87.16%	96.54%
Frequency	91.79%	93.07%	81.83%	85.65%	96.11%
Markov	74.66%	71.83%	75.31%	67.65%	69.25%
BiLSTM	86.23%	75.55%	96.46%	92.81%	96.95%
vBiLSTM	90.98%	96.31%	93.24%	90.13%	95.46%
API-Bert	84.71%	93.29%	91.87%	81.49%	92.73%
API-SIF	88.24%	96.91%	88.89%	87.83%	95.59%
DMalNet	82.46%	96.12%	87.33%	85.09%	95.03%
CruParamer	90.98%	96.31%	93.24%	90.13%	95.46%
API2Vec	**99.24%**	**99.49%**	**99.25%**	**99.15%**	**99.41%**

is robust to concept drift. This is mainly owing to the graph model. More concretely, although new variants of malware show different syntax from the previous one, they are behaviorally identical. Since our graph model exhibits well invariance regardless of the order of API calls, it could reveal these behavior accurately. Consequently, the generated paths representing the behaviors of a new variant are highly similar to those of the previous malware.

7.3.2 Malware of New Type. This type involves malware from a new type that was not appeared in the training dataset. We use one type of malware as the testing set, and other types for training. The numbers of testing malware are as follows, `virus` (984), `backdoor` (1,675), `worm` (2,464), `grayware` (3,904), `downloader` (4,193).

As reported in Table 4, a model performs differently for various types of malware. E.g., DeepWalk achieves 93.39% of *f1-score* for `worm`, while only 77.72% for `virus`. In contrast, API2Vec performs well on all these types, i.e., larger than 99%. The reason is as follows. Although different types of malware exhibit various behaviors, they share many common fine-grained operations, e.g., downloading files, remote control, etc. Compared to existing methods, API2Vec can capture these fine-grained behaviors more accurately with the graph model and heuristic random walk. Thus, it provides better generalization ability when detecting malware of new types than other methods, and thus exhibits well robustness to concept drift.

7.4 Robustness to Adversarial Attacks (RQ3)

Recently, several adversarial attacks are proposed against malware detection models [33, 59]. We employ the attack method from [33] to perturb malware samples in the testing dataset. The method generates API sequences pieces and inserts them into an original malware sequence, m, to produce an adversarial sequence, m^*, which aims to minimize the predicted malicious probability on m^*

Table 5: Performance on detecting adversarial samples.

Model	rate	Model	rate	Model	rate
Word2Vec	62.49%	Frequency	70.79%	API-Bert	49.00%
DeepWalk	67.13%	Markov	70.06%	API-SIF	67.63%
Node2VecB	66.47%	BiLSTM	55.16%	DMalNet	71.77%
Node2VecD	66.54%	vBiLSTM	59.90%	CruParamer	75.86%
API2Vec	**97.38%**				

Table 6: Performance of various learning models.

Model	Word2Vec			Doc2Vec		
	precision	recall	f1-score	precision	recall	f1-score
SVM	87.33%	89.08%	88.20%	99.16%	99.25%	99.20%
RF	96.75%	92.08%	94.36%	**99.47%**	99.04%	99.26%
k-NN	94.70%	94.29%	94.49%	99.23%	**99.84%**	**99.54%**

for recurrent neural network (RNN) models. Finally, we obtain 4,396 adversarial samples and predict them with the trained detectors.

Let detection rate be the ratio of the number of successfully detected adversarial examples to the total number (i.e., 4,396), higher detection rate denotes better robustness. As shown in Table 5, API2Vec shows better robustness to adversarial attacks, with a detection rate of 97.38%. In other words, of 4,396 adversarial malware samples, API2Vec successfully identifies 4,281, which is 1,330, 1,856, and 1,126 more than DeepWalk, BiLSTM, and DMalNet, respectively. We contribute the improvement to the fine-grained behaviors extracted from the graph model, which are relatively invariant. Specifically, the adversarial attack perturbs the raw sequences by carefully inserting API calls to alter its syntax and semantics. However, the proposed random walk algorithm in API2Vec employs many heuristic rules, which are related to the number of nodes, number of edges, and logical time. Therefore, the generated paths, which represent fine-grained behavior, are not easily comprised unless a large amount of API calls are inserted, which yet violates the principle of minimal perturbations. DMalNet also utilizes graph to model the API call relationship. However, it mainly focuses on the sequential execution of APIs and discards the inter-process relationship, leading to inferior performance compared to API2Vec.

7.5 Ablation Study (RQ4)

7.5.1 The Gain of Proposed Components. The key components of API2Vec are the graph model and the random walk algorithm following three categories of heuristic rules. Thus, we setup the following models to measure the gain of proposed methods. 1) Sequence, the baseline method that performs Doc2Vec on raw sequences directly. 2) Graph, it uses our graphs yet is configured with naïve random walk. 3) Graph+Affinity (G^*), it walks with 2 affinity rules, which are basic rules to guide the direction. 4) G^*+Coverage, it extends G^* with 2 coverage-oriented rules. 5) G^*+Behavior, it extends G^* with 5 behavior-oriented rules. 6) API2Vec, it uses all rules.

As reported in Figure 5, each component brings a performance boost, i.e., the graph model imposes 0.67% improvement, and the heuristic rules improve 4.08% further, in *f1-score*. Specifically, compared to Sequence (94.79%), Graph achieves 95.46%, implying that the proposed graph model brings 0.67% improvement. With two

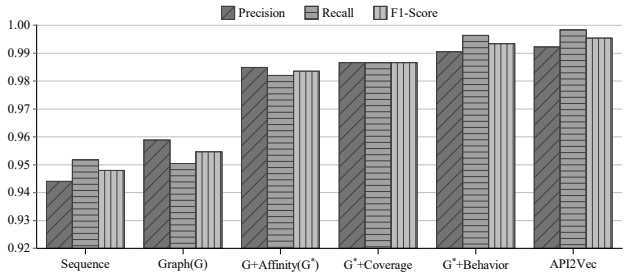


Figure 5: Performance of proposed components.

basic rules, Graph+Affinity improves Graph by 2.87%, proving the effectiveness of heuristic rules upon walk. Furthermore, when coverage- or behavior-oriented rules are added, *f1-score* is further improved by 0.29% and 0.99%, respectively. Note that behavior-oriented rules contribute more, because they aim to reveal the intra- and inter-process behavior more accurately while coverage-oriented rules seek to generate more diverse paths. Finally, with all rules, API2Vec reaches up to 99.54% of *f1-score*, achieving a total of 4.75% improvement compared to the baseline model.

7.5.2 Various Learning Models. API2Vec uses Doc2Vec to pre-train the path embeddings, and employs SVM, RF and *k*-NN to train the malware detector. Hence, we compare Doc2Vec against the commonly used Word2Vec model upon embedding, and evaluate the detector when cooperated with various ML models. As shown in Table 6, Doc2Vec performs much better than Word2Vec, e.g., 99.54% compared to 94.49% in *f1-score*. This is because Doc2Vec encodes not only APIs but also paths which is helpful to represent the concept of paths, while in contrast, Word2Vec only encodes APIs. In addition, the three ML models with Doc2Vec perform well and closely, e.g., 99.20%, 99.26%, and 99.54% in *f1-score*, respectively, demonstrating the effectiveness of the proposed path embedding.

7.6 Runtime Overhead (RQ5)

The time overhead of API2Vec comes from four parts, i.e., graph building, path generation, API embedding, and detection. We perform the experiments on a Ubuntu(20.04.2 LTS) server with 22-core Intel(R) Xeon(R) CPU E5-2690 2.60GHz and 300 GBs of RAM. The average processing time per sequence is 77.56ms, 107.76ms, 62.35ms, and 0.2ms respectively for the four parts, taking a total time of 247.87ms for one sequence. In comparison, it takes 71ms, 71ms, 35.7ms, 101.6ms, 19ms, 187.4ms, 223.4ms, and 198.35ms to process one sequence for Frequency, Markov, BiLSTM, vBiLSTM, API-Bert, API-SIF, DMalNet, and CruParamer, respectively. Although API2Vec takes relatively more time, the 247.87ms detection time is relatively modest considering that it typically takes 2 minutes to obtain the API call sequence through sandbox. Thus, we believe that API2Vec is feasible for malware detection in practical scenarios.

7.7 Case Studies

We provide two examples of real-world malware that are detected by API2Vec but not by other models to illustrate its effectiveness.

7.7.1 Sample 1 [4]. The main process M sets hooks and drops several executables, from which two processes, A and B, are launched. A probes and gathers environmental information, while B downloads payload from a remote site. Although the two processes execute independently, their API calls are highly interleaved. E.g., a call VirtualAllocEx of B is logically preceded by CreateMutexW and followed by CreateRemoteThread of B, but is now interfered by multiple RegQueryValueExW of A. Thus, existing methods fail to accurately identify intra-process behavior of B. API2Vec overcomes this limitation by grouping together APIs of the same process, and linking M and A, M and B, respectively. This enables successful revelation of the behaviors between processes from generated paths. E.g., it can capture a path from the setting of hooks by M to information collection of and data storing performed by A.

7.7.2 Sample 2 [5]. The main process M creates two processes A and B, where A further creates a child process A_1. A probes the environments and then clears A_1 after it has enumerated the modules in memory. B simply drop several executables and involves only a few API calls. Existing methods fail to detect the sample mainly due to the long sequence issued by A_1, which consists of multiple repetitive sequences, e.g., several calls of ReadProcessMemory (accounting for 64.5% of the sequence) followed by WriteFile. API2Vec overcomes this challenge by designing coverage-oriented rules. This approach allows for more paths to be generated, thereby exposing diverse behaviors, without being limited by the constraints imposed by a single node (e.g., ReadProcessMemory in this sample).

8 RELATED WORK

Many studies perform malware analysis on sequences of API calls. Ki et al. [38] view the raw sequence of API calls as DNA of malware, and then employ DNA sequence comparison to extract the API call pattern. These patterns help detect currently known malware and discover previously unknown malware. Daht et al. [19] divide the system API call sequence into triples with n-grams, and then use logistic regression and shallow neural network for malware classification. Hansen et al. [32] convert the malware sequence into a feature vector, mainly including APIs, the API frequency. Then, they leverage information entropy on the feature vector to classify malware. Kim et at. [39] use n-gram and TF-IDF to generate features of system calls, and employ SVM for malware detection.

Many recent studies employ DL in malware detection. Rosenberg et al. [63] divide the raw API sequence into several sub-sequences and detect each of them with RNN respectively. Chen et al. [17] represent the API calls by the API along with varying degrees of sensitivity. And, on the encoded sequences of API calls, malware classifiers are trained with TextCNN and Bi-LSTM. Yesir et al. [74] present API-Bert, which employs Fasttext and BERT to learn the embeddings of APIs. The experimental results show that DL model performs well in malware detection.

Instead of directly operating raw sequences, some studies represent API calls with graphs and then performing analysis with graph algorithms. E.g., Zhang et al. [76] build API relationship Graph by parsing the Android API documents and design knowledge graph embedding algorithm to learn the representation of APIs, which are applied to malware detection models. Elhadi, et al. [24] extracts the data dependency relationship from the raw sequence and build API graph. Then, they employ longest common subsequence to perform graph matching, which is further used for malware detection. Ding, et al. [21] use the data dependency to build a behavior graph and then detect malware with graph matching.

Our method is inspired by these studies. However, it is different in many aspects. First, our graph model consists of two graphs to characterize intra and inter-process behavior respectively. Meanwhile, we assign edges with temporal attributes, which are important to describe the process behavior. In contrast, existing methods build graphs with the spatial relationship on APIs. Second, we design several heuristic rules upon walking, so that the paths can reveal diverse fine-grained behaviors. Although DeepWalk and Node2Vec design random walk algorithms, they are more focused on the structural information of graphs and are not suitable for our temporal graph. Third, we encode these paths instead of APIs, which is different from many DL-based methods that directly employ Word2Vec or Bert to generate embeddings of APIs.

9 LIMITATION AND DISCUSSION

We conducted API2Vec three independent times and obtained a total of 16 false negatives (FN). We manually investigated them and summarized following reasons. 1) Repeated benign paths (9 FN). The presence of multiple edges between neighboring APIs results in a number of repeated paths during random walk. Meanwhile, these paths represent benign operations. Thus, benign semantics contribute more in generating embeddings, thereby evading detection. 2) Many processes with simple, repeated actions (2 FN). Similar to the first reason, this results in a stronger emphasis on benign semantics in the embeddings, making it harder to detect. We suspect that these 11 samples employ evasion techniques by incorporating meaningless API calls. 3) Short and simple sequences (5 FN). These samples have short sequences, i.e., less than 50, far less than the average length of 412. Of them, 2 samples terminated early before exhibiting further malicious behavior and 3 used neutral APIs but passed with sensitive parameters. We will explore techniques (e.g., excluding redundant paths, using run-time parameters) to address these problems in the future.

10 CONCLUSION

This paper presents API2Vec, a graph based API embedding method for malware detection. The main advantages of API2Vec over existing embedding methods lie in two aspects. That is, the graph model is invariant and is able to capture the intra- and inter-process behaviors more concisely and accurately. Meanwhile, the heuristic random walk algorithm follows the temporal order of nodes with several behavior and coverage-oriented rules. Therefore, it is able to mine many diverse and fine-grained behaviors. The results show that API2Vec outperforms state-of-the-arts in malware detection, and is robust to adversarial attacks and concept drifts.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (61972392, 6207245). Yuede Ji is supported by University of North Texas. We would like to express our deepest gratitude to Xiaohui Chen, Yiran Zhu, and Runhan Song for their invaluable comments and assistance throughout our experiments.

REFERENCES

[1] 2022. Cuckoo Sandbox. Website. https://cuckoosandbox.org/.

[2] 2022. NIST: National Institute of Standards and Technology. Website. https://www.nist.gov/.

[3] 2022. VirusSign Incorporation. Website. https://www.virussign.com.

[4] 2023. VirusTotal reports. Website. https://www.virustotal.com/gui/file/bd49943e2db92d09287923b159b111c33f3374344cf39109aebaa81fc2cd16e0.

[5] 2023. VirusTotal reports. Website. https://www.virustotal.com/gui/file/d9ba27e5554c95350ca0800051d6eca19251c3a8d98662b2b7afbc06e8ead9a6.

[6] Muhamed Fauzi Bin Abbas and Thambipillai Srikanthan. 2017. Low-complexity signature-based malware detection for IoT devices. In *International Conference on Applications and Techniques in Information Security*. Springer, 181–189. https://doi.org/10.1007/978-981-10-5421-1_15

[7] Faraz Ahmed, Haider Hameed, M Zubair Shafiq, and Muddassar Farooq. 2009. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*. 55–62. https://doi.org/10.1145/1654988.1655003

[8] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018). https://doi.org/10.48550/arXiv.1808.01400

[9] Eslam Amer and Ivan Zelinka. 2020. A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence. *Computers & Security* 92 (2020), 101760. https://doi.org/10.1016/j.cose.2020.101760

[10] Eslam Amer, Ivan Zelinka, and Shaker El-Sappagh. 2021. A Multi-Perspective malware detection approach through behavioral fusion of API call sequence. *Computers & Security* 110 (2021), 102449. https://doi.org/10.1016/j.cose.2021.102449

[11] H. S. Anderson and P. Roth. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints* (April 2018). https://doi.org/10.48550/arXiv.1804.04637 arXiv:1804.04637 [cs.CR]

[12] Ömer Aslan Aslan and Refik Samet. 2020. A comprehensive review on malware detection approaches. *IEEE Access* 8 (2020), 6249–6271. https://doi.org/10.1109/ACCESS.2019.2963724

[13] Seyyed Mojtaba Bidoki, Saeed Jalili, and Asghar Tajoddin. 2017. PbMMD: A novel policy based multi-process malware detection. *Engineering Applications of Artificial Intelligence* 60 (2017), 57–70. https://doi.org/10.1016/j.engappai.2016.12.008

[14] Ross Brewer. 2016. Ransomware attacks: detection, prevention and cure. *Network Security* 2016, 9 (2016), 5–9. https://doi.org/10.1016/S1353-4858(16)30086-1

[15] Ferhat Ozgur Catak, Javed Ahmed, Kevser Sahinbas, and Zahid Hussain Khand. 2021. Data augmentation based malware detection using convolutional neural networks. *PeerJ Computer Science* 7 (Jan. 2021), e346. https://doi.org/10.7717/peerj-cs.346

[16] Patrick Shicheng Chen, Shu-Chiung Lin, and Chien-Hsing Sun. 2015. Simple and effective method for detecting abnormal internet behaviors of mobile devices. *Information Sciences* 321 (2015), 193–204. https://doi.org/10.1016/j.ins.2015.04.035

[17] Xiaohui Chen, Zhiyu Hao, Lun Li, Lei Cui, Yiran Zhu, Zhenquan Ding, and Yongji Liu. 2022. CruParamer: Learning on Parameter-Augmented API Sequences for Malware Detection. *IEEE Transactions on Information Forensics and Security* 17 (2022), 788–803. https://doi.org/10.1109/TIFS.2022.3152360

[18] John Cloonan. 2019. Advanced Malware Detection – Signatures vs. Behavior Analysis. https://www.cyberdefensemagazine.com/advanced-malware-detection/

[19] George E Dahl, Jack W Stokes, et al. 2013. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 3422–3426. https://doi.org/10.1109/ICASSP.2013.6638293

[20] 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 *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489. https://doi.org/10.1109/SP.2019.00003

[21] Yuxin Ding, Xiaoling Xia, Sheng Chen, and Ye Li. 2018. A malware detection method based on family behavior graph. *Computers & Security* 73 (2018), 73–86. https://doi.org/10.1016/j.cose.2017.10.007

[22] Daniele Cono D'Elia, Lorenzo Invidia, and Leonardo Querzoni. 2021. Rope: Covert multi-process malware execution with return-oriented programming. In *European Symposium on Research in Computer Security*. 197–217. https://doi.org/10.1007/978-3-030-88418-5_10

[23] Mohamed El Boujnouni, Mohamed Jedra, and Noureddine Zahid. 2015. New malware detection framework based on N-grams and support vector domain description. In *2015 11th international conference on information assurance and security (IAS)*. IEEE, 123–128. https://doi.org/10.1109/ISIAS.2015.7492756

[24] Ammar Ahmed E Elhadi, Mohd Aizaini Maarof, and Bazara IA Barry. 2013. Improving the detection of malware behaviour using simplified data dependent API call graph. *International Journal of Security and Its Applications* 7, 5 (2013), 29–42. https://doi.org/10.14257/ijsia.2013.7.5.03

[25] Lejun Fan, Yuanzhuo Wang, Xueqi Cheng, Jinming Li, and Shuyuan Jin. 2015. Privacy theft malware multi-process collaboration analysis. *Security and Communication Networks* 8, 1 (2015), 51–67. https://doi.org/10.1002/sec.705

[26] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. 2018. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Transactions on Information Forensics and Security* 13, 8 (2018), 1890–1905. https://doi.org/10.1109/TIFS.2018.2806891

[27] Ivan Firdausi, Alva Erwin, Anto Satriyo Nugroho, et al. 2010. Analysis of machine learning techniques used in behavior-based malware detection. In *2010 second international conference on advances in computing, control, and telecommunication technologies*. IEEE, 201–203. https://doi.org/10.1109/ACT.2010.33

[28] Fabio De Gaspari, Dorjan Hitaj, Giulio Pagnotta, Lorenzo De Carli, and Luigi V Mancini. 2020. The naked sun: Malicious cooperation between benign-looking processes. In *International Conference on Applied Cryptography and Network Security*. Springer, 254–274. https://doi.org/10.1007/978-3-030-57878-7_13

[29] Daniel Gibert, Carles Mateu, and Jordi Planes. 2020. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications* 153 (2020), 102526. https://doi.org/10.1016/j.jnca.2019.102526

[30] Google. 2021. Colab. Website. https://colab.research.google.com/.

[31] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864. https://doi.org/10.1145/2939672.2939754

[32] Steven Strandlund Hansen, Thor Mark Tampus Larsen, et al. 2016. An approach for detection and family classification of malware based on behavioral analysis. In *2016 ICNC*. IEEE, 1–5. https://doi.org/10.1109/ICCNC.2016.7440587

[33] Weiwei Hu and Ying Tan. 2018. Black-box attacks against RNN based malware detection algorithms. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*. https://doi.org/10.48550/arXiv.1705.08131

[34] Kyriakos K. Ispoglou and Mathias Payer. 2016. MalWASH: Washing Malware to Evade Dynamic Analysis. In *Proceedings of the 10th USENIX Conference on Offensive Technologies (WOOT'16)*. 106–117. https://doi.org/10.5555/3027019.3027029

[35] Yuede Ji, Yukun He, Dewei Zhu, Qiang Li, and Dong Guo. 2014. A mulitiprocess mechanism of evading behavior-based bot detection approaches. In *International conference on information security practice and experience*. Springer, 75–89. https://doi.org/10.1007/978-3-319-06320-1_7

[36] 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 *26th USENIX Security Symposium (USENIX Security 17)*. 625–642. https://doi.org/10.5555/3241189.3241239

[37] Kaggle. 2021. Malware Dataset. Website. https://www.kaggle.com/datasets/blackarcher/malware-dataset.

[38] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. 2015. A novel approach to detect malware based on API call sequence analysis. *International Journal of Distributed Sensor Networks* 11, 6 (2015), 659101. https://doi.org/10.1155/2015/659101

[39] Chan Woo Kim. 2018. Ntmaldetect: A machine learning approach to malware detection using native api system calls. *arXiv preprint arXiv:1802.05412* (2018). https://doi.org/10.48550/arXiv.1802.05412

[40] Dhilung Kirat and Giovanni Vigna. 2015. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *ACM SIGSAC Conference on Computer and Communications Security*. 769–780. https://doi.org/10.1145/2810103.2813642

[41] Engin Kirda and Christopher Krügel. 2006. Behavior-based Spyware Detection. In *USENIX Security Symposium*.

[42] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. 2009. Effective and efficient malware detection at the end host.. In *USENIX security symposium*, Vol. 4. 351–366.

[43] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. 2021. Does every second count? time-based evolution of malware behavior in sandboxes. In *Proceedings of the Network and Distributed System Security Symposium, NDSS. The Internet Society*. https://doi.org/10.14722/ndss.2021.24475

[44] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[45] Ce Li, Zijun Cheng, He Zhu, Leiqi Wang, Qiujian Lv, Yan Wang, Ning Li, and Degang Sun. 2022. DMalNet: Dynamic malware analysis based on API feature engineering and graph learning. *Computers & Security* 122 (2022), 102872. https://doi.org/10.1016/j.cose.2022.102872

[46] Ce Li, Qiujian Lv, Ning Li, Yan Wang, Degang Sun, and Yuanyuan Qiao. 2022. A novel deep framework for dynamic malware detection based on API sequence intrinsic features. *Computers & Security* 116 (2022), 102686. https://doi.org/10.1016/j.cose.2022.102686

[47] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-An, and Heng Ye. 2018. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3216–3225. https://doi.org/10.1109/TII.2017.2789219

[48] Olav Lysne. 2018. *The Huawei and Snowden Questions: Can Electronic Equipment from Untrusted Vendors be Verified? Can an Untrusted Vendor Build Trust Into*

Electronic Equipment? Springer Nature. https://doi.org/10.1007/978-3-319-74950-1

[49] Microsoft. 2015. Microsoft Malware Classification Challenge. Website. https://www.kaggle.com/competitions/malware-classification/data.

[50] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013). https://doi.org/10.48550/arXiv.1301.3781

[51] Fahad Mira. 2019. A review paper of malware detection using api call sequences. In *2019 2nd International Conference on Computer Applications & Information Security (ICCAIS)*. IEEE, 1–6. https://doi.org/10.1109/CAIS.2019.8769564

[52] Robert Moir. 2009. Defining Malware: FAQ. Website. https://docs.microsoft.com/en-us/previous-versions/tn-archive/dd632948(v=technet.10)?redirectedfrom=MSDN.

[53] Bruce Ndibanje, Ki Hwan Kim, Young Jin Kang, Hyun Ho Kim, Tae Yong Kim, and Hoon Jae Lee. 2019. Cross-method-based analysis and classification of malicious behavior by api calls extraction. *Applied Sciences* 9, 2 (2019), 239. https://doi.org/10.3390/app9020239

[54] Quoc-Dung Ngo, Huy-Trung Nguyen, Van-Hoang Le, and Doan-Hieu Nguyen. 2020. A survey of IoT malware and detection methods based on static features. *ICT Express* 6, 4 (2020), 280–286. https://doi.org/10.1016/j.icte.2020.04.005

[55] Stavros D Nikolopoulos and Iosif Polenakis. 2017. A graph-based model for malware detection and classification using system-call groups. *Journal of Computer Virology and Hacking Techniques* 13, 1 (2017), 29–46. https://doi.org/10.1007/s11416-016-0267-1

[56] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. 2019. Dynamic malware analysis in the modern era—A state of the art survey. *ACM Computing Surveys (CSUR)* 52, 5 (2019), 1–48. https://doi.org/10.1145/3329786

[57] Abdurrahman Pektaş and Tankut Acarman. 2020. Deep learning for effective Android malware detection using API call graph embeddings. *Soft Computing* 24, 2 (2020), 1027–1043. https://doi.org/10.1007/s00500-019-03940-5

[58] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*. 729–746. https://doi.org/10.5555/3361338.3361389

[59] Xiaowei Peng, Hequn Xian, Qian Lu, and Xiuqing Lu. 2021. Semantics aware adversarial malware examples generation for black-box attacks. *Applied Soft Computing* 109 (2021), 107506. https://doi.org/10.1016/j.asoc.2021.107506

[60] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710. https://doi.org/10.1145/2623330.2623732

[61] Marco Ramilli, Matt Bishop, and Shining Sun. 2011. Multiprocess malware. *2011 6th International Conference on Malicious and Unwanted Software* (2011), 8–13. https://doi.org/10.1109/MALWARE.2011.6112320

[62] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. 2008. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 108–125. https://doi.org/10.1007/978-3-540-70542-0_6

[63] Ishai Rosenberg, Asaf Shabtai, et al. 2018. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 490–510. https://doi.org/10.1007/978-3-030-00470-5_23

[64] Raj Samani. 2021. McAfee Labs Threats Report–June 2021. *McAfee Labs* (2021).

[65] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. 2016. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing* 15, 1 (2016), 83–97. https://doi.org/10.1109/TDSC.2016.2536605

[66] Virus Total. 2022. Virustotal-free online virus, malware and url scanner. *Online: https://www.virustotal.com/en* 2 (2022).

[67] Trung Kien Tran and Hiroshi Sato. 2017. NLP-based approaches for malware classification from API sequences. In *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*. IEEE, 101–105. https://doi.org/10.1109/IESYS.2017.8233569

[68] Swapna Vemparala, Fabio Di Troia, Visaggio Aaron Corrado, Thomas H Austin, and Mark Stamo. 2016. Malware detection using dynamic birthmarks. In *Proceedings of the 2016 ACM on international workshop on security and privacy analytics*. 41–46. https://doi.org/10.1145/2875475.2875476

[69] Deepak Venugopal and Guoning Hu. 2008. Efficient signature based malware detection on mobile devices. *Mobile Information Systems* 4, 1 (2008), 33–49.

[70] Peng Wang, Zhijie Tang, and Junfeng Wang. 2021. A novel few-shot malware classification approach for unknown family recognition with multi-prototype modeling. *Computers & Security* 106 (2021), 102273. https://doi.org/10.1016/j.cose.2021.102273

[71] 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 (EuroS&P)*. IEEE, 47–62. https://doi.org/10.1109/EuroSP.2019.00014

[72] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. 2017. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)* 50, 3 (2017), 1–40. https://doi.org/10.1145/3073559

[73] Suleiman Y Yerima, Sakir Sezer, and Gavin McWilliams. 2014. Analysis of Bayesian classification-based approaches for Android malware detection. *IET Information Security* 8, 1 (2014), 25–36. https://doi.org/10.1049/iet-ifs.2013.0095

[74] Salih Yesir and İbrahim Soğukpinar. 2021. Malware Detection and Classification Using fastText and BERT. In *2021 9th International Symposium on Digital Forensics and Security (ISDFS)*. IEEE, 1–6. https://doi.org/10.1109/ISDFS52919.2021.9486377

[75] Zenodo. 2019. Dynamic Malware Analysis kernel and user-level calls. Website. https://zenodo.org/record/1203289.

[76] 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*. 757–770. https://doi.org/10.1145/3372297.3417291

[77] Zhaoqi Zhang, Panpan Qi, and Wei Wang. 2020. Dynamic malware analysis with feature engineering and feature learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1210–1217. https://doi.org/10.1609/aaai.v34i01.5474

Received 2023-02-16; accepted 2023-05-03