Contextual Weisfeiler-Lehman Graph Kernel For Malware Detection

Annamalai Narayanan, Guozhu Meng, Liu Yang, Jinliang Liu and Lihui Chen
Nanyang Technological University, Singapore.
annamala002@e.ntu.edu.sg, {gzmeng, yangliu}@ntu.edu.sg, liuj0081@e.ntu.edu.sg, elhchen@ntu.edu.sg

Abstract—In this paper, we propose a novel graph kernel specifically to address a challenging problem in the field of cyber-security, namely, malware detection. Previous research has revealed the following: (1) Graph representations of programs are ideally suited for malware detection as they are robust against several attacks, (2) Besides capturing topological neighbourhoods (i.e., structural information) from these graphs it is important to capture the context under which the neighbourhoods are reachable to accurately detect malicious neighbourhoods.

We observe that state-of-the-art graph kernels, such as Weisfeiler-Lehman kernel (WLK) capture the structural information well but fail to capture contextual information. To address this, we develop the Contextual Weisfeiler-Lehman kernel (CWLK) which is capable of capturing both these types of information. We show that for the malware detection problem, CWLK is more expressive and hence more accurate than WLK while maintaining comparable efficiency. Through our large-scale experiments with more than 50,000 real-world Android apps, we demonstrate that CWLK outperforms two state-of-the-art graph kernels (including WLK) and three malware detection techniques by more than 5.27% and 4.87% F-measure, respectively, while maintaining high efficiency. This high accuracy and efficiency make CWLK suitable for large-scale real-world malware detection.

keywords — graph kernels, malware detection, program analysis

I. INTRODUCTION

Malware detection has evolved as one of the challenging problems in the field of cyber-security as the attackers continuously enhance the sophistication of malware to evade novel detection techniques. Malware for various platforms such as desktop and mobile devices is growing at an alarming rate. For instance, Kaspersky reports [1] detecting 4 million malware infections in 2015 which is a 216% increase over 2014. This volume and growth rate clearly highlights an imperative need for automated malware detection solutions.

To perform automated malware detection, security analysts resort to program analysis and machine learning (ML) techniques. Typically, this process involves extracting semantic features from suitable representations of programs (e.g., assembly code, call graphs) and detecting malicious code or behavior patterns using ML classifiers [2]–[7].

A major reason for such tremendous growth rate in malware is the production of malware variants. Typically, the attackers produce large number of variants of the same malware by resorting to techniques such as variable renaming and junk code insertion [3, 4, 7, 8]. These variants perform same malicious functionality, with apparently different syntax, thus evading syntax-based detectors. However, higher level semantic representations such as call graphs, control- and data-flow graphs, control-, data- and program-dependency graphs mostly stay similar even when the code is considerably altered [3, 5, 7, 8]. In this work, we use a common term, Program Representation Graph (PRG) to refer to any of these aforementioned graphs. As PRGs are resilient against variants, many works in the past have used them to perform malware detection. In essence, such works cast malware detection as a graph classification problem and apply existing graph mining and classification techniques [8]. Some methods such as [2, 3, 5] note that ML classifiers are readily applicable on data represented as vectors and attempt to encode PRGs as feature vectors. Typically, these techniques face two challenges:

• (C1) Expressiveness. PRGs are complex and expressive data structures that characterize topological relationships among program entities. Representing them as vectors is a non-trivial task. In many cases vectorial representations of PRGs fail to capture all the vital information. For instance, AppContext [2], a well-known Android malware detection approach represents apps as PRGs and ends up capturing features from individual nodes without their topological neighbourhood information. With such loss of expressiveness, attacks that span across multiple PRG nodes could not be effectively detected.

• (C2) Efficiency. The scale of malware detection problem is such that we have millions of samples already and thousands streaming in every day. Many classic graph mining based approaches (e.g., [8]) are NP hard and have severe scalability issues, making them impractical for real-world malware detection [3, 9].

Graph Kernels. One of the increasingly popular approaches in ML for graph-structured data is the use of graph kernels. Recently, efficient and expressive graph kernels such as [11]–[14] have been proposed and widely adopted in many application areas (e.g, bio- and chemo-informatics [15]–[18]). Some of them support explicit feature vector representations of graphs (e.g, [11]–[13]). Thus both the aforementioned challenges C1 and C2 are effectively addressed by these graph kernels. Therefore, it just suffices to use a graph kernel together with a kernelized ML classifier (e.g., SVM) and we have a scalable, effective and ready-to-use malware detector. Recently, three approaches [3], [6] and [9], have successfully demonstrated using these general purpose graph kernels for malware detection.

Research Gap. However, a major problem in using these general purpose graph kernels on PRGs is that, they are not designed to take domain-specific observations into account.
For instance, recent research on malware analysis has revealed that besides capturing neighbourhood (i.e., structural) information from PRGs it is important to capture the context under which the neighbourhoods are reachable to accurately detect malicious neighbourhoods [2, 7] (explained in detail in §II). Many existing graph kernels such as [11] and [12] can capture and compare structural information from PRGs effectively. However, they are not designed to capture the reachability context, as it is a strong domain-specific requirement and hence fail to do so. To address this, we develop a novel graph kernel which is capable of capturing both the aforementioned types of information.

For similar domain-specific reasons, researchers from other fields such as computer vision [18], bio- and chemoinformatics [15]–[17] have developed a number of kernels that specifically suit their applications. Despite graphs being natural representations of programs and amenable for various activities, the program analysis research community has not devoted significant attention to development of domain-specific graph kernels. We take the first step towards this, by developing a kernel on PRGs which specifically suits our task of malware detection.

Our approach. To improve the accuracy of malware detection process, we propose a method to enrich the feature space of a graph kernel that inherently captures structural information with contextual information. We apply this feature-enrichment idea on a state-of-the-art graph kernel, namely, Weisfeiler-Lehman kernel (WLK) [11] to obtain the Contextual Weisfeiler-Lehman kernel (CWLK). Specifically, CWLK associates to each sub-structure feature of WLK a piece of information about the context under which the sub-structure is reachable in the course of execution of the program. A sub-structure appearing in two different PRGs will match only if it is reachable under the same context in both PRGs. We show that for the malware detection problem, CWLK is more expressive and hence more accurate than WLK and other state-of-the-art kernels while maintaining comparable efficiency.

Experiments. Through our large-scale experiments with more than 50,000 Android apps, we demonstrate that CWLK outperforms two state-of-the-art graph kernels and three malware detection solutions in terms of accuracy, while maintaining high efficiency (§V).

We make an efficient implementation of the proposed kernel (along with the dataset information) publicly available.

II. BACKGROUND & MOTIVATION

In this section, we motivate the design of our kernel by describing why considering just the structural information from PRGs is insufficient to determine the maliciousness of a sample and how supplementing it with contextual information helps to increase detection accuracy. To this end we use a real-world Android malware2 from the Geinimi family which steals users’ private information. We contrast its behavior with that of a well-known benign app, Yahoo Weather.

Geinimi’s execution. The app is launched through a background event such as receiving a SMS or call. Once launched, it reads the user’s personal information such as geographic location and contacts and leaks the same to a remote server. The (simplified) malicious code portion pertaining to the location information leak is shown in Fig. 1 (a). The method leak_location reads the geographic location through getLatitude and getLongitude Application Programming Interfaces (APIs). Subsequently, it calls leak_info_to_url method to leak the location details (through DataOutputStream.writeBytes) to a specific server. The Data Dependency Graph (DDG) corresponding to the code snippet is shown in Fig. 1 (b). The nodes in DDG are labeled with the sensitive APIs that they invoke.

Yahoo Weather’s execution. On the other hand, Yahoo Weather could be launched only by user’s interaction with the device (e.g., by clicking the app’s icon on the dash board). The app then reads the user’s location and sends the same to its weather server to retrieve location-specific weather predictions. Hence, DDG portions of Yahoo Weather is same as that of Geinimi.

Contextual information. From the explanations above, it is clear that both the apps leak the same information in the same fashion. However, what makes Geinimi malicious is the fact that its leak happens without the user’s consent. In other words, unlike Yahoo Weather, Geinimi leaks private information through an event which is not triggered by user’s

---

1https://sites.google.com/site/cwlkernel

2SHA256:056200032fa2abed5beb428b5e5d58ff5aa8115019736013d87f442032b6bc
interaction. We refer to this as a leak happening in user-unaware context. On the same lines, we refer to Yahoo Weather's leak as happening in user-aware context.

As explained in [7] and [2], in the case of Android apps, one could determine whether a PRG node is reachable under user-aware or user-unaware context by examining its entry point nodes. Following this procedure we add the context as an attribute to every DDG node. This context annotated DDG of Geinimi and Yahoo Weather are shown in Fig. 1 (c) and (d), respectively.

**Requirements for effective detection.** From the aforementioned example the two requirements that makes a malware detection process effective can be identified:

**R1) Capturing structural information.** Since malicious behaviors often span across multiple nodes in PRGs, just considering individual nodes (and their attributes) in isolation is not enough. Capturing the structural (i.e., neighborhood) information from PRGs is of paramount importance.

**R2) Capturing contextual information.** Considering just the structural information without the context is not enough to determine whether a sensitive behavior is triggered with or without user’s knowledge. For instance, if structural information alone is considered, the features of both Geinimi and Yahoo Weather apps become identical, thus making the latter a false positive. Hence, it is important for the detection process to capture the contextual information as well to make the detection process more accurate.

Many existing graph kernels could address the first requirement well. However, the second requirement which is more domain-specific makes the problem particularly challenging. To the best of our knowledge, none of the existing graph kernels support capturing this reachability context information along with structural information. Hence, this gives us a clear motivation to develop a new kernel that specifically addresses our two-fold requirement.

### III. Definitions and Notations

The formal definitions and notations that will be used throughout the paper are presented in this section.

**Definition 1 (Program Representation Graph).** \(PRG = (N, E, \lambda, \xi)\) is a directed graph where \(N\) is a set of nodes and each node \(n \in N\) denotes program entity such as a function or instruction. \(E \subseteq (N \times N)\) is a set of edges and each edge \(e(n_1, n_2) \in E\) denotes either control- or data-flow or dependency from \(n_1\) to \(n_2\). \(\lambda\) is the set of labels that characterize the (security-sensitive) operations of a node and \(\xi : N \rightarrow \lambda\) is a labeling function which assigns a label to each node. \(\xi\) is a set of events that denote the context of a node and \(C : N \rightarrow \xi\) is a function which assigns the context to each node.

**Definition 2 (Context).** The context of a node \(n \in N\) in the PRG of a program \(P\) is a set of attributes \(\xi(n)\) that govern the reachability of \(n\) in the course of execution of \(P\).

**Examples of contexts.** In the case of Windows executables, the guard conditions that govern the execution of a node could be considered as its context. Unlike Windows (and other desktop OS) binaries, Android and iOS mobile apps typically have multiple entry points [2, 7]. Hence, in the case of such mobile apps, besides guard conditions, the categories of entry points through which a node is reachable could also be considered as its context. Similar platform-specific constraints and observations could be considered while defining the contexts for executables of other platforms.

### IV. Contextual Weisfeiler-Lehman Graph Kernel

In this section, we begin by explaining how the regular WLK can be applied to perform malware detection using PRGs and how it falls short. Subsequently, we introduce our CWLK and discuss how it addresses the shortcomings of WLK. Finally, we prove CWLK's semi-definitiveness and analyze its time complexity.

**A. Regular Weisfeiler-Lehman Graph Kernel**

WLK computes the similarities between graphs based on the 1-dimensional WL test of isomorphism [11].

**WL test of isomorphism.** Suppose we are to determine whether a given pair of graphs \(G\) and \(G'\) are isomorphic. The WL test of isomorphism works by augmenting the node labels by the sorted set of labels of neighboring nodes. This process is referred to as label-enrichment and new labels are referred as neighborhood labels. Thus, in each iteration \(i\) of the WL algorithm, for each node \(n \in N\), we get a new neighborhood label, \(\lambda_i(n)\) that encompass the \(i^{th}\) degree neighborhood around \(n\). \(\lambda_i(n)\) could be optionally compressed using a hash function \(f : \Sigma^* \rightarrow \Sigma\) such that \(f(\lambda_i(n)) = f(\lambda_i(n'))\), iff \(\lambda_i(n) = \lambda_i(n')\). To test graph isomorphism, the relabeling process is repeated until the neighborhood label sets of \(G\) and \(G'\) differ, or the number of iterations reaches a specific threshold. Therefore, one iteration of WL relabeling is equivalent to a function \(r((N, E, \lambda_i)) = (N, E, \lambda_{i+1})\) that transforms all graphs in the same manner.

**Definition 3 (WL sequence).** Define the WL graph at height \(i\) of the graph \(G = (N, E, \lambda)\) as the graph \(G_i = (V, E, \lambda_i)\). The sequence of graphs

\[
G_0, G_1, ..., G_h = (V, E, \lambda_0), (V, E, \lambda_1), ..., (V, E, \lambda_h)
\]

(i.e., \(\lambda_0 = \lambda\)) is called the WL sequence up to height \(h\) of \(G\), where \(G_0 = G\) is the original graph and \(G_i = r(G_{i-1})\) is the graph resulting from the first relabeling, and so on.

**Definition 4 (WL kernel).** Given a valid kernel \(k(., .)\) and the WL sequence of graph of a pair of graphs \(G\) and \(G'\), the WL graph kernel with \(h\) iterations is defined as

\[
k^{(h)}_{WL}(G, G') = k(G_0, G'_0) + ... + k(G_h, G'_h)
\]

where \(h\) is the number of WL iterations and \(G_0, G'_1, ..., G'_h\) are the WL sequences of \(G\) and \(G'\), respectively. \(h\) is referred as height of the kernel.

Intuitively, WLK counts the common neighborhood labels in two graphs. Hence we have \(k^{(h)}_{WL}(G, G') = |\{\lambda_i(n) | \lambda_i(n')\}\|\) iff \(f(\lambda_i(n)) = f(\lambda_i(n'))\) for \(i \in \{1, ..., h\}\), \(n \in N, n' \in N'\), where \(f\) is injective and the sets \(\{f(\lambda_i(n)) | n \in N \cup N'\}\) and \(\{f(\lambda_i(n)) | n \in N \cup N'\}\) are disjoint for all \(i \neq j\).

**Example & WLK’s shortcoming.** We now apply WLK on the real-world examples discussed in §II to see if it distinguishes malicious and benign neighborhoods clearly, facilitating accurate detection. For the ease of illustration,
Algorithm 1 CWLK - Contextual re-labeling

Input: 
\[ G = (N, E, \lambda_0, \xi) \] — PRG with set of nodes (N), set of edges (E) and set of node labels (\(\lambda_0\)) and context for each node (\(\xi\))
\(h\) — number of iterations

Output: 
\[ \{G_0, G_1, \ldots, G_h\} \] - contextual WL sequence of height \(h\)

1: procedure CONTEXTUAL-RE-LABEL(G, \(h\))
2: \:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\
ϕσ(h)(G′) for G′. Then,
\[ kσ(h)(G, G′) = ϕσ(h)(G), ϕσ(h)(G′) = |\{(σ(i(n)), σ(i(n′)))\} \]
σ(i) = σ(i′), i \in \{1, ..., h\}, n, n′ \in Σ∗ \]
(4)

Summing over all σ from the vocabulary Σ∗, we get
\[ k_{\text{CWL}}^{(h)}(G, G′) = \sum_{σ \in Σ∗} k_{\text{CWL}}^{(h)}(G, G′) = \sum_{σ \in Σ∗} ϕσ(h)(G)ϕσ(h)(G′) \]
\[ = |\{(σ(i(n)), σ(i(n′)))\}σ(n) = σ(n′), i \in \{1, ..., h\}, n, n′ \in Σ∗ \} \]
\[ = |\{(σ(i(n)), σ(i(n′)))|f(σ(n)) = f(σ(n′)), \]
\[ i \in \{1, ..., h\}, n, n′ \in Σ∗ \}| \]
(5)

where the last equality follows from the fact that f is injective. As f(σ) ≠ f(σ′) if σ ≠ σ′, the string σ corresponds to exactly one contextual neighborhood label and k_{\text{CWL}}^{(h)} defines a kernel with corresponding feature map ϕ_{\text{CWL}}(h), such that
\[ ϕ_{\text{CWL}}(h) = (ϕ_{\text{CWL}}(h))(G) = \sigma \in Σ∗ \]
(6)

Complexity. The runtime complexity of CWLK with h iterations on a graph with n nodes and e edges is O(he) (assuming that e > n) which is same as that of WLK. More specifically, the neighborhood label computation with sorting operations (lines 10-12 of Algorithm 1) take O(e) time for one iteration and the same for h iterations take O(he). The inclusion of context (lines 6-8, 13-15), does not incur additional overhead as e >> |Σ|. Hence the final time complexity remains as O(he). For a detailed derivation and analysis of the time complexity of WLK, we refer the reader to [11].

Efficient computation of CWLK on K graphs. When computing CWLK on K graphs to obtain K × K kernel matrix, a naive approach would involve K² comparisons, resulting a time complexity of O(K²he). However, as mentioned in [11], a Bag-of-Features (BoF) model based optimization could be performed to arrive the kernel matrix in O(Khe + K²hn) time. This optimized computation involves the following steps: (1) A vocabulary Σ of all the contextual neighborhood labels of nodes across the K graphs is obtained in O(Khe) time. This facilitates representing each of the K graphs as feature vectors of |Σ| dimensions. (2) Subsequently, K × K kernel matrix can be computed by multiplying these vectors in O(K²hn) time. In summary, CWLK has the same efficiency as that of WLK and supports explicit feature vector representations of PRGs.

Relation to other spatial contextual kernels. Two recently proposed graph kernels [13] and [14], consider incorporating the spatial context information to neighborhood subgraph features. They define context of a subgraph feature as another subgraph appearing in its vicinity. As mentioned earlier, in our malware detection problem we refer to attributes of a node which determine its reachability as its context. This reachability context is different from spatial context discussed in [13] and [14]. Hence CWLK is concomitantly different from these two kernels.

### Table I: Composition of dataset

| Portion of dataset | Dataset source                  | # of samples |
|--------------------|---------------------------------|--------------|
| Malware portion    | DREBIN [4], Virus Share [19]    | 29877        |
| Benign portion     | Google Play [20]                | 25000        |

### Table II: Dataset Statistics

|                  | CG (Avg ± Std) | ICFG (Avg ± Std) |
|------------------|----------------|------------------|
|                   | # of Nodes     | # of Edges       | # of Nodes     | # of Edges       |
| 1556 ± 998       | 3327 ± 1803    | 15323 ± 9844     | 22745 ± 20922  |

### V. Evaluation

We conducted large scale experiments involving more than 50,000 Android apps from two real-world malware datasets to evaluate the accuracy and efficiency of CWLK. We compare CWLK’s performance against that of two state-of-the-art kernels and three Android malware detection solutions.

#### A. Datasets

DREBIN [4] provides a collection on 5,560 Android malware apps collected from 2010 to 2012. More recently, Virusshare [19] released a collection of 24,317 malware apps collected from 2010 to 2014. We combined these two datasets and use them in our evaluation. For the benign portion of the dataset, we collected 25,000 benign top-selling apps from Google Play store [20] that were released around the same time. Thus, our dataset contains a total of 54,877 apps. The composition of our dataset is presented in Table I.

#### Graph Representations considered

As mentioned in §I the proposed CWLK could be applied on any type of PRGs to perform malware detection. In our evaluations, we experimented with two types of PRGs namely, call graphs (CGs) and inter-procedural control-flow graphs (ICFGs). The nodes of a CG represent the methods present in an app; its directed edges represent the calling/called relations between the methods. The nodes of an ICFG represent the individual instructions present in those methods; its edges represent the control-flows among those instructions. These two types of PRGs are chosen since they capture program semantics at different levels of granularities. The statistics on the average number of nodes and edges of the PRGs in our dataset are presented in Table II. Details on construction of these PRGs are presented later in §V-B.

#### Training and Test sets

60% of the samples were randomly chosen from the datasets and used for training the classifier and the remaining 40% samples are used to test their performances. The classifiers’ hyper-parameters are determined on the training set using 5-fold cross-validation, whereas the test set is only used for determining the final detection performance. We repeat this procedure 5 times and average the results.

#### B. Implementation & Comparative Analysis

PRG construction & Context Identification. Both CG and ICFG of the apps in our datasets are constructed through static analysis using Soot [21], a well-known Android static analysis workbench. The nodes are labeled with the security-sensitive APIs they access and are annotated with context information. Nodes that do not access any sensitive APIs are removed. We use the category of entry points of each of the nodes in these graphs as their contexts. The procedure proposed in DroidSIFT [7] is used to identify and categorize each entry point as being in ‘user-aware’ or ‘user-unaware’ context.

Comparison with Graph Kernels. CWLK’s accuracy and efficiency is compared against those of WLK [11] and Neighborhood Hash Graph Kernel (NHGK) [12]. Since these kernels cannot capture context information, we use the comparative
analysis against them to ascertain whether including context information significantly improves the accuracy without affecting the efficiency. We implemented all these kernels in about 2,170 lines of Python code. For all the kernels, BoF model based implementation similar to the one discussed in § IV-B is used obtain explicit feature vector representation of samples. **Comparison with Malware Detection Solutions.** Also, our approach is compared against three light-weight state-of-the-art ML based Android malware detection solutions, namely, DREBIN [4], Allix et al. [5] and ADAGIO [3]. To this end, we re-implemented DREBIN and Allix et al.’s approaches through consultations with the authors. For ADAGIO, an open-source implementation provided by the authors is used. Since the malware detection accuracy of these solutions predominantly depend on the features they use, we briefly introduce them here. DREBIN [4] considers features such as sensitive Android APIs, permissions and components used by apps. Allix et al. [5] constructs the CFGs of individual methods and represent them as signature strings which are subsequently used as features. ADAGIO [3] constructs CGs and uses byte-code instructions to assign labels to nodes. NHGK [12] is used to extract CG neighborhoods as features and a histogram-intersection (HI) kernel SVM is trained to detect malware. Due to severe scalability issues (explained in § V-D3) ADAGIO is ran only once on our dataset. All other techniques are ran 5 times and the average results are reported.

**C. Experimental Design**

**Research Questions.** Through our evaluations, we seek to address the following research questions: (1) Does including the context information in PRGs through CWLK significantly improve the malware detection accuracy? (2) Does capturing context information incur significant computation overhead to adversely affect the efficiency? (3) Does our context-based detection approach outperform existing malware detection solutions? Two separate experiments are designed to address these questions.

**Experiment E1.** In order to evaluate the first two questions, the following experiment is conducted: First, the CGs of all the training set apps are constructed. Then, the WLK, NHGK and CWLK kernels are applied on the CGs to obtain respective kernel matrices. Subsequently, a SVM classifier is trained with each of these kernels to detect malicious apps. Finally, the test set apps are subjected to the aforementioned kernel computation processes and are used to evaluate the models. The same procedure is repeated with ICFGs.

**Experiment E2.** In order to address question (3), we subject DREBIN [4], Allix et al. [5] and ADAGIO [3] solutions to the same training and test sets. We compare them against the best performing model that uses CWLK (obtained from experiment E1).

**Setting the parameter h:** We experimented with different kernel heights \( h = 0 \) to 5 for CWLK (see eq. (3)). The average number of contextual neighborhood features (from 5 runs of E1) for different values of \( h \) on the two PRGs is reported in Fig. 2. It is evident that the number of features does not increase significantly after \( h = 2 \) on both PRGs. This is because we have removed nodes that do not access sensitive APIs which affects the connectivity and restricts the neighborhood sizes. In other words, we seldom have neighborhoods around nodes that span beyond degree 2. Similar trend is observed in WLK and NHGK features. Hence we restrict the height \( h \) to be 0, 1 and 2 for all three kernels. Thus, for each kernel applied on each PRG, we have three SVM classifiers (one for each \( h \)). Therefore we have a total of 18 malware detection models under comparison (see Table III for details).

**Evaluation metrics.** Standard evaluation metrics such as Precision, Recall and F-measure are used to determine the effectiveness of malware detection [5]. Efficiency is determined in terms of training and testing durations.

**Evaluation Setup.** All the experiments were conducted on Intel Xeon Hexa-core E5-2640 processor (2.50 GHz) with 32 GB RAM running Ubuntu 14.04.

**D. Results and Discussions**

1) **Impact of context information on F-measure:** We compare the Precision, Recall and F-measure of the malware detection process through CWLK with those of WLK and NHGK. This is to ascertain whether incorporating the context information in PRGs boosts the effectiveness of detection. These results for the 18 models are presented in Table III, from which the following inferences are drawn:

- At the outset, two general observations are made: (1) All models perform better on ICFGs compared to CGs. This is because, ICFG is a more fine-grained representation of programs than CG, which enables it to capture program semantics at a finer level and thereby boosting the detecting accuracy. Hence, we conclude that in our experiments ICFG is a more effective representation for malware detection. (2) Considering larger neighborhoods helps capturing the structural information better which in turn reflects in better performances. This is evident as Precision, Recall and F-measure values get better with increasing values of \( h \) for all the three kernels. However, this observation may not hold for large values of \( h \), as nodes that are far apart will be considered for neighborhood re-labeling, leading to a noisy re-labeling process.

- It is clear that CWLK outperforms both WLK and NHGK in terms of F-measure consistently on both CG and ICFG representations.

- **Since the only difference between WLK and CWLK is the latter’s capability to capture the context information, evidently this is the reason for CWLK’s superior performance.**

- Also, CWLK achieves better Precision than WLK in all the experiments, consistently. This indicates that CWLK suffers
Table III: Comparison of CWLK against state-of-the-art Graph Kernels (metrics expressed as % values) - avg. over 5 runs

| PRG Type | Kernel | h = 0 | Precision | Recall | F-measure | Precision | Recall | F-measure | Precision | Recall | F-measure |
|----------|--------|-------|-----------|--------|-----------|-----------|--------|-----------|-----------|--------|-----------|
| CG       | WLK[11] | 89.03 | 75.75     | 85.00  | 85.93     | 87.28     | 86.15  | ±0.78    | 86.71     | ±0.88  |
|          | NHGK[12]| 63.05 | 61.14     | 61.56  | 67.18     | 69.02     | 74.17  | ±1.11    | 72.14     | ±1.88  |
|          | CWLK    | 79.27 | 79.85     | 88.50  | 87.98     | 89.23     | 90.10  | ±0.96    | 89.64     | ±0.94  |
| ICFG     | WLK[11] | 85.49 | 83.59     | 87.84  | 87.48     | 90.17     | 90.17  | ±0.92    | 90.45     | ±0.90  |
|          | NHGK[12]| 68.68 | 70.73     | 75.78  | 81.01     | 78.11     | 82.28  | ±1.50    | 80.14     | ±1.29  |
|          | CWLK    | 89.74 | 81.25     | 96.77  | 92.08     | 94.36     | 97.15  | ±0.28    | 95.82     | ±0.66  |

Impact of context information on efficiency

Experimentally, we conclude that CWLK through its virtue of capturing both structural and contextual information, significantly improves upon the accuracy of WLK for the malware detection task without hurting the efficiency.

CWLK Vs. state-of-the-art malware detectors

We now compare CWLK based detection with the state-of-the-art Android malware detection solutions to study whether contextual PRG neighborhoods makes good features for malware detection, through experiment E2. For CWLK based detection, ICFG representation with h = 2 is used as it offers the best performance. The Precision, Recall and F-measures of each of these methods are reported in Table IV. The following observations are made from the table:

- Table IV: Comparison of CWLK based malware detection against the state-of-the-art detectors (avg. over 5 runs)

| Method               | Precision | Recall | F-measure |
|----------------------|-----------|--------|-----------|
| Drebin[4]            | 97.02     | ±0.93  | 85.60     | ±1.03     | 90.95     | ±0.71     |
| Allix et al.[5]      | 88.24     | ±0.74  | 86.29     | ±1.66     | 87.25     | ±0.54     |
| Adago[3]             | 92.18     | ±0.55  | 87.52     | ±0.53     | 89.68     | ±0.86     |
| CWLK                 | 97.15     | ±0.28  | 94.53     | ±0.75     | 95.82     | ±0.66     |
Clearly, CWLK based malware detection outperforms all the compared solutions in terms of F-measure. In particular, our approach outperforms the best performing technique (i.e., DREBIN) by 4.87% F-measure. In terms of Precision, our approach outperforms ADAGIO and Allix et al.’s methods and is comparable to DREBIN. In terms of Recall, ours outperforms other methods.

Out of the methods compared, DREBIN does not use both structural and contextual features. ADAGIO and Allix et al.’s approaches use structural information but not contextual information. This reveals that capturing both these types of information is the reason for our approach’s superior performance, reinforcing our findings from experiment E1. We now compare the efficiency of CWLK based detection against that of state-of-the-art malware detectors. It is noted that these techniques use different features and classifiers and hence a wide variation in training and testing durations is expected. The results of this comparison are presented in Fig. 4, from which the following observations are made:

- DREBIN being a light-weight non PRG based approach it has significantly higher efficiency than all other methods, including ours.
- Allix et al.’s method is similar to ours in terms of using PRG based features. Hence our efficiency is comparable to this method.
- ADAGIO uses NHGK and HI kernel SVM in the primal formulation. Hence it takes a prohibitively long time for training and testing. Our method is far more efficient than ADAGIO.

In conclusion, our method’s efficiency is comparable to that of other PRG based methods, far better than heavy-weight approaches and inferior to non PRG based light-weight methods.

VII. ACKNOWLEDGMENT

We thank the authors of [4] and [5], for their suggestions that helped us re-implement their methods.

REFERENCES

[1] Kaspersky 2014 Annual Threat Report. URL. https://securelist.com/files/2015/12/Kaspersky-Security-Bulletin-2015_FINAL_EN.pdf
[2] Yang, Wei, et al. “Appcontext: Differentiating malicious and benign mobile app behaviors using context.” Proc. of the International Conference on Software Engineering (ICSE). 2015.
[3] Gascon, Hugo, et al. “Structural detection of android malware using embedded call graphs.” Proceedings of the 2013 ACM workshop on Artificial intelligence and security. ACM, 2013.
[4] Arp, Daniel, et al. “Drebin: Effective and explainable detection of android malware in your pocket.” Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS), 2014.
[5] Allix, Kevin, et al. “Empirical assessment of machine learning-based malware detectors for Android.” Empirical Software Engineering (2014): 1-29.
[6] Sahs, Justin, and Latifur Khan. "A machine learning approach to android malware detection." Intelligence and Security Informatics Conference (EISIC), 2012 European. IEEE, 2012.
[7] Zhang, Mu, et al. "Semantics-aware Android malware classification using weighted contextual API dependency graphs.” Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014.
[8] Fredriksson, Matt, et al. “Synthesizing near-optimal malware specifications from suspicious behaviors.” Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010.
[9] Anderson, Blake, et al. "Graph-based malware detection using dynamic analysis.” Journal in Computer Virology 7.4 (2011): 247-258.
[10] Shervashidze, Nino, et al. "Graph kernels.” The Journal of Machine Learning Research 11 (2010): 1201-1242.
[11] Shervashidze, Nino, et al. "Weisfeiler-leman graph kernels.” The Journal of Machine Learning Research 12 (2011): 2539-2561.
[12] Hido, Shohie, and Hisashi Kashima. "A linear-time graph kernel.” Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on. IEEE, 2009.
[13] Navarin, N., Sperduti, A., & Tesselli, R. (2015, November). Extending local features with contextual information in graph kernels. In Neural Information Processing (pp. 271-279). Springer International Publishing.
[14] Costa, Fabrizio, and Kurt De Grave. "Fast neighborhood subgraph pairwise distance kernel.” Proceedings of the 26th International Conference on Machine Learning. Omnipress, 2010.
[15] Borgwardt, Karsten M., et al. "Protein function prediction via graph kernels.” Bioinformatics 21.suppl 1 (2005): i47-i56.
[16] Frohlich, Holger, Jörg K. Wegner, and Andreas Zell. Assignment kernels for chemical compounds.” Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on. Vol. 2. IEE, 2005.
[17] Mahé, Pierre, and Jean-Philippe Vert. "Graph kernels based on tree patterns for molecules.” Machine learning 75.1 (2009): 3-35.
[18] Harchaoui, Zaid, and Francis Bach. "Image classification with segmentation graph kernels.” Computer Vision and Pattern Recognition, 2007. CVPR’07. IEEE Conference on. IEEE, 2007.
[19] Virus Share malware dataset. URL. http://virusshare.com
[20] Google Play Store. URL. https://play.google.com/store
[21] Soot framework. URL. https://sable.github.io/soot